SCALABLE ENERGY-EFFICIENT MICROARCHITECTURES WITH COMPUTATIONAL ERROR TOLERANCE

A Dissertation
Presented to
The Academic Faculty

By

Bobin Deng

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2021

SCALABLE ENERGY-EFFICIENT MICROARCHITECTURES WITH COMPUTATIONAL ERROR TOLERANCE

Approved by:

Dr. Thomas M. Conte, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Alexandros Daglis
School of Computer Science
Georgia Institute of Technology

Dr. Arijit Raychowdhury
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Jeanine Cook
Computer Science Research Institute
Sandia National Laboratories

Date Approved: April 22, 2021

Enjoy a grander sight by climbing to a greater height

Zhihuan Wang

ACKNOWLEDGEMENTS

Thanks to my family for giving me unlimited tolerance and encouragement in the past few years. Their support gave me continuous motivation to complete my Ph.D. degree.

Thanks to my thesis advisor, Dr. Thomas M. Conte. He has given me great support in all aspects, including research work, writing, presentation, and teaching skills, so that I can successfully complete my degree.

Thanks to Dr. Erik Debenedictis and Dr. Jeanine Cook for giving me many support and suggestions on research projects, which helped me quickly enter a new research domain and gave very constructive feedback on my research.

Thanks to all the members of TINKER and CRNCH, through continuous learning and discussion with each other, my research capability and knowledge have been significantly improved.

Thanks to thesis committee members, Dr. Thomas M. Conte, Dr. Hyesoon Kim, Dr. Alexandros Daglis, Dr. Arijit Raychowdhury and Dr. Jeanine Cook, for their time to give very valuable feedback on my thesis and research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Dennard scaling of conventional semiconductor technology has reached its limit resulting in issues pertaining to leakage current and threshold voltage. Energy-savings found at the transistor level by simply lowering supply voltage are no longer available for these devices (e.g., MOSFETs) and has reached the Landauer-Shannon limit. Recent proposals of minivolt switch technologies aim to extend the technology scaling roadmap by maintaining a high on/off ratio of drain current with a much lower supply voltage. However, high intermittent error probabilities in millivolt switches constraints their V_{dd} reduction for traditional architectures. Thus, there is an urgent need for scalable and energy-efficient micro-architectures with computational error-tolerance.

This thesis systematically leverages the error detection and correction properties of the Redundant Residue Number System (RRNS) by varying the number of non-redundant (n) and redundant (r) components (residues), and selects and discusses trade-offs about configuration points from a two-dimensional *(n, r)-RRNS* design plane that meet certain capabilities of error detection and/or correction. Being able to efficiently handle resilience in this *(n, r)-RRNS* plane significantly improves reliability, allowing further V_{dd} reduction and energy savings.

First, the necessary implementation details of RRNS cores are discussed. Second, scalable RRNS micro-architectures that simultaneously support both error-correction and checkpointing with restart capabilities for uncorrectable errors are proposed. Third, novel RRNS-based adaptive checkpointing&restart mechanisms are designed that automatically guarantee reliability while minimizing the energy-delay product (EDP). Finally, the RRNS design space is explored to find the optimal (n, r) configuration points. For similar reliability when compared to a conventional binary core (running at high V_{dd}) without computational error tolerance, the proposed RRNS scalable micro-architecture reduces EDP by 53% on average for memory-intensive workloads and by 67% on average for non-memory-

intensive workloads.

This thesis's second topic is to alleviate fault rate and power consumption issues of exascale computing. Faults in High-Performance Computing (HPC) have become an urgent challenge with estimated Mean Time Between Failures (MTBF) of exascale system projected as only several minutes with contemporary methodologies. Unfortunately, existing error-tolerance technologies in the context of HPC systems have serious deficiencies such as insufficient error-tolerance coverage, high power consumption, and difficult integration with existing workloads. Considering Department of Energy (DOE) guidelines that limit exascale power consumption to 20 MW, this thesis highlights the issue of energy usage and proposes a thread-level fault tolerance mechanism compatible with current state-of-the art exascale programming models while simultaneously meeting the requirements of full system error protection. Additionally, an efficient micro-architecture and corresponding mechanisms that can support thread level RRNS are discussed. Experimental results show that this strategy reduces energy consumption by 62.25% and the Energy-Delay-Product by 58.67% on average when compared with state-of-the-art black box resilience techniques.

CHAPTER 1
INTRODUCTION AND BACKGROUND

Dennard scaling [1] has been one of the critical technology roadmap trendings through past decades for computer efficiency improvement. This scaling's main idea is that transistors consume the same amount of power per unit area as they scale down in size. However, leakage current and threshold voltage issues from smaller MOSFET devices caused Dennard scaling to be terminated [2]. This termination essentially negates possible performance benefits that Moore's law may provide.

This has not reached the limit of irreversible logic operations. Landauer [3] presented the lower bound of energy dissipation of an irreversible logic operation to be on the order of kT, where k is the Boltzmann's constant and T is the absolute temperature. However, MOSFET is constrained by the $50\times$ higher Landauer-Shannon limit [4]. The reason for this disparity is that MOSFETs are limited to a sub-threshold slope of $\frac{kT}{q}ln10 \approx 60$ mV/decade. Certain new devices, known as millivolt switches, such as Tunnel FETs [5] and Ferroelectric FETs [6], are possible to break through this wall. However, as the signal strength of a cell phone becomes weaker as it moves further from the transmitter tower, Agarwal et al. [7] conclude that error probability increases exponentially as signal energy is lowered: $P_e = exp(-\frac{E_{signal}}{kT})$. P_e in this equation denotes error probability, and E_{signal} represents the signal energy of this transistor. If the size of a new device continually scaling, it eventually leads to an error because there is no enough energy to distinguish the '0' or '1' status.

Therefore, to keep the failure rate of millivolt switch-based microarchitectures within bounds and achieve optimal or suboptimal energy reduction, error-tolerance consideration must be incorporated. In other words, to punch through the existing power wall via millivolt switches, efficient error-detection, correction, and/or checkpoint&rollback schemes

are required. The technology roadmap would then be possible to scale further by reducing the switch signal energy to close the Landauer limit.

1.1 Computational Error Tolerance

Standard error-correcting codes (ECC) [8] has already widely been using in modern communication systems and storage units. However, *ECC lacks the theoretical capability to protect the computational units and corresponding components, such as Arithmetic Logic Unit (ALU) or accumulator*. Therefore, traditional ECC is insufficient to protect the whole computing system at a low V_{dd}. Computational error-tolerant techniques that are commonly used are now described in the following subsections.

1.1.1 Dual/Triple Modular Redundancy

Dual or Triple Modular Redundancy (D/TMR) is a conventional computational error-tolerant scheme [9]. From the perspective of error-tolerant capability, DMR is able to detect one error while TMR can correct one error from one of the components. As shown in Figure 1.1,the basic idea of TMR is to make two additional copies of the existing computational logic and then followed by a majority vote from these three outputs. If one of the computational components arises an error while the remaining two are correct, this error is easily fixed via an error-free majority vote. While TMR is simple and straightforward to implement, its overhead is more than 200% in both component area and power to detect and correct the single error. The energy-saving potentials from lowering V_{dd} would therefore be eliminated.

1.1.2 Residue Checking

IBM commercial processor series, such as z990, z10, z196, POWER6 and POWER7 [10, 11, 12, 13] use residue checking scheme to achieve the error tolerance. A residue is the remainder from a given number is divided by a predefined module. The residue check-

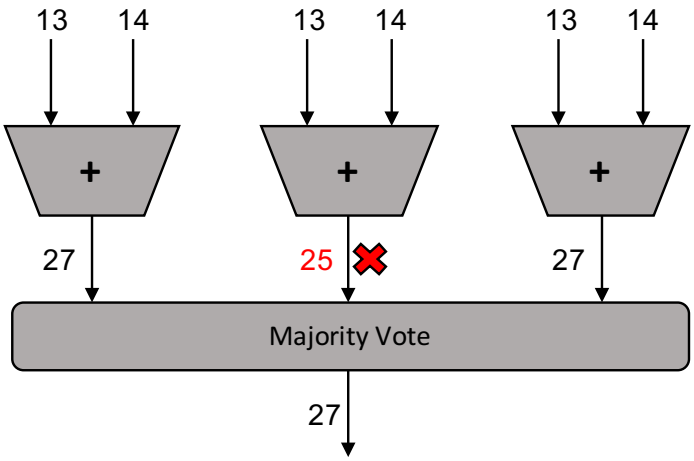


Figure 1.1: Triple Modular Redundancy in action.

ing methodology consists of two independent datapaths followed by a comparator, and leverages the closure property of residue arithmetic wherein $(A\%m\ op\ B\%m)\%m = (A\ op\ B)\%m$, for integers A, B, modulus m and arithmetic operation op. An error is detected whenever the comparator returns a not-equal output. From the flowchart in Figure 1.2, m usually is represented by a small integer, such as 3, 9, or 15 in IBM processor series.

Generally speaking, the overhead of residue checking technologies' one-time error detection is low. This is mainly because the bit-width of its redundant datapath is constrained by the selected modulus, which is usually less or equal to 4 bits. Unfortunately, this lightweight protection may negatively impact the full system reliability. For example, for $m = 5$, $\frac{1}{5}$ of possible errors are undetectable (say the correct value is 2, all erroneous values v satisfying $v\%5 = 2$ such as 7, 12, 17 etc. would remain undetectable).

Furthermore, an error detection operation must be performed right after every arithmetic computation. If no residue error check is inserted after the corresponding arithmetic operation, then this error generated during the computation becomes undetectable forever. Finally, as the residue checking datapath can only detect errors generated during the current arithmetic computation, the input values must be verified (such as via performing an ECC check on every register read) before this computation.

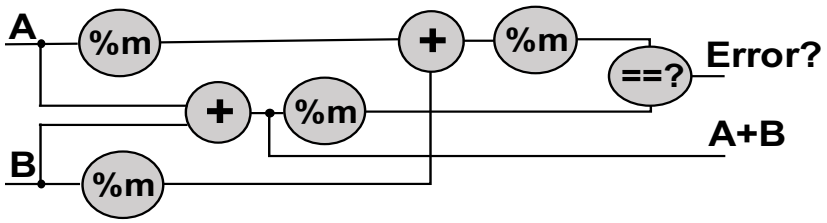


Figure 1.2: Residue Checking Flowchart

An improved computational error-tolerant approach (RRNS, described below) overhauls residue checking to eliminate the above issues while still maintaining excellent system robustness. RRNS has the following benefits:

1. RRNS may significantly reduce the undetectable error rate in general.

2. RRNS does not require checking the output of every arithmetic operation.

3. RRNS does not require checking input values.

Furthermore, ALU optimizations become feasible in RRNS, resulting in significantly more energy-efficient addition, subtraction, and multiplication. Before discussing RRNS, the Residue Number Systems (RNS) needs to be first described.

1.1.3 Residue Number Systems (RNS)

Residue Number Systems (RNS) have been widely utilized in many popular application domains such as digital signal processing (DSP) [14, 15, 16], cryptography (RSA) [17, 18, 19] and in-memory neural network acceleration [20]. Due to the carry-free property [21, 22] and small bit-width datapaths, RNS is able to get benefits via replacing the conventional binary system in computationally intensive domains. In an n-RNS, a non-negative integer $X < M$ bijectively maps to a set of n smaller integers, known as residues. Given a set of n co-prime bases or moduli $B = \{m_i \in \mathbb{N}, i = 1, 2, 3, ..., n\}$, the i^{th} residue is defined as $|X|_{m_i} = X \ \% \ m_i$. The RNS range of representable numbers is euqal to $M = \prod_{i=1}^{n} m_i$.

The arithmetic operations of RNS, such as addition, subtraction, and multiplication, are very efficient. This is mainly because each residue performs parallel and independent arithmetic computations and does not interfere with and interact with each other. The parallel and independent properties offer great potential to improve the performance and energy reduction, particularly for multiplication which is always slow and power-hungry with conventional binary design. However, the general RNS division operation requires relatively high overhead algorithms [23, 24], and most RNS architectures avoid division-intensive workloads. For workloads with only a small ratio of division instructions, besides the specific RNS division algorithm, RNS can also be converted to binary to perform a binary division and then convert the output from binary back to RNS format after the binary division completes.

1.1.4 Redundant Residue Number Systems (RRNS)

AS the residue has independence and isolation properties, the error generated on a particular residue should not propagate to others. By adding r redundant moduli to an n-RNS system, error tolerance can then be achieved. This is known as the Redundant Residue Number System, or *(n,r)-RRNS* in this case and now has an extended set of bases/moduli: $B = \{m_i \in \mathbb{N}, i = 1, 2, 3, ..., n, n+1, ..., n+r\}$. Any non-negative integer number smaller than M $(= \prod_{i=1}^{n} m_i)$ can still be represented uniquely by its original n non-redundant residues while the remaining r residues store redundant information to support error tolerance.

Since errors in a particular residue are self-contained, these errors from a particular reisue should not infect the others. Furthermore, the errant residue can be detected and/or corrected via the remaining residues [25]. Table 1.1 provides a simple *(4,2)-RRNS* system ($n = 4$, $r = 2$) example with a toy base set (3, 5, 2, 7, 11, 13) to explain how RRNS works for arithmetic computation. Table 1.1 is extended from the example table in [26]. Decimal $10 \equiv (2, 4, 1, 1, 7, 3)$ in this example and the range $M = 3 \times 5 \times 2 \times 7 = 210$.

Table 1.1: A *(4, 2)-RRNS* example with a toy base set (3, 5, 2, 7, 11, 13). 11 and 13 are the redundant bases.

Decimal	%3	%5	%2	%7	%11	%13
10	1	0	0	3	10	10
19	1	4	1	5	8	6
10+19=29	(1+1)%3=2	4	1	1	7	3
10*19=190	(1*1)%3=1	0	0	1	3	8

In practice, base sets with more realistic ranges, such as (211, 421, 256, 347, 503, 509), are discussed in Section 2.9. A *(4,2)-RRNS* system has the capability of correcting a one residue in error (One Error Correction;1EC) *or* detecting two residues in error (Two Error Detection;2ED) [25]. The relationships between (n,r) and RRNS error tolerance capabilities with other related details are discussed in Chapter 2.2 and 2.3.

In general, RRNS is a great error-tolerant methodology with energy-efficient and low area overhead potentials. Based on these advantages, RRNS can be widely used in remote embedded systems, high-performance computers, spacecraft design, etc., all of which have specific requirements of fault tolerance and low energy consumption. This thesis analyzes RRNS from bit-level parallelism for millivolt switch and thread-level parallelism for exascale computing. It also proposes related microarchitectures to achieve error-tolerance and energy-efficiency goals.

1.2 Contributions

While modeling the RRNS cores with some empirical configuration points may get more energy-efficient than unprotected binary ones with similar reliability, a systematic *(n, r)-RRNS* design exploration yields even more energy benefit and is able to find out the corresponding optimal or suboptimal configurations. On the one hand, by thoroughly exploiting the potential of RRNS reliability, millivolt switches' signal energy can further approach the theoretical Landauer limit. On the other hand, by applying the RRNS methodology at thread level granularity, the two urgent challenges of exascale computing, fault tolerance

and high energy consumption, can be alleviated to a certain extent.

This thesis starts with a specific RRNS configuration(zero-dimension, (4,2)-RRNS) and verifies the feasibility and the potential of this resilient methodology. And then enhancing the zero-dimensional design to a two-dimensional *scalable* scalable architecture framework supporting c-residue error correction and d-residue error detection (cECdED). In theory, c and d can be any non-negative integers. For example, 1EC is equivalent to 1EC0ED, and 2ED is represented as 0EC2ED. The term 'scalable' above means that the number of RRNS non-redundant (n) and redundant (r) moduli can be extended with error tolerance adjustment. As such, the following key contributions are made for microarchitecture designs of millivolt switches:

1. Propose a scalable, energy-efficient RRNS-based microarchitecture that simultaneously supports error correction and checkpointing&restart, when used with upcoming millivolt switches.

2. Design and analysis of an efficient RRNS multiplier unit using the index-sum technique.

3. Design a set of novel RRNS-based checkpointing&restart mechanisms that benefit from optimized checkpoint size and adaptive time interval adjustment. They narrow the signal energy gap further to close the theoretical Landauer limit. Moreover, compared with other checkpointing technologies, these methodologies further improve memory usage efficiency.

4. Perform a systematic design space exploration of (n,r)-RRNS-cECdCR microarchitectures via a tradeoff analysis between reliability and energy-delay overheads of error correction and checkpointing. Solving such a discrete convex optimization problem results in finding out the best RRNS configurations that minimize energy-delay-product (EDP) while maximizing reliability.

The bit-level RRNS evaluation results with reasonable (n,r)-RRNS configurations (more details are available in Chapter 2.3) present three insights: (i) Cores that employ a *hybrid* detection-correction strategy (i.e., $cd > 0$) are more resilient. However, they suffer from the combined overhead of both mechanisms in terms of energy as well as performance. (ii) For memory-intensive workloads, the EDP of *correction-only* systems ($d = 0$) are superior to *detection-only* ($c = 0$) ones within a reasonable range, resulting in 53% EDP reduction. This is because of the higher state overheads in checkpoint-based systems for memory-intensive workloads. (iii) For non-memory-intensive workloads, detection-based systems are superior, resulting in 67% EDP reduction. Unless otherwise mentioned, all results in this topic are compared against a conventional binary core that is not computationally error-tolerant (runs at high V_{dd}) of similar reliability[1].

In the optimization of exascale computing, the fault-tolerant RRNS API is added to the Habanero C/C++ library (HClib)[27], which provides easy-to-use interfaces for the programming model of exascale computing. Moreover, the thread-level RRNS CPU microarchitecture is proposed. When applications that require high fault-tolerance requirements in a heterogeneous exascale computing system, they can be assigned to these fault-tolerant cores to execute and achieve the purpose of fault-tolerance. Compared with the traditional Triple Modular Redundancy(TMR) method, RRNS has the advantage of lower energy consumption. Therefore, the following contributions are made for thread-level RRNS design:

1. Propose a thread-level Redundant Residue Number System (RRNS) scheme and design the corresponding microarchitecture by following the unique execution mode of thread-level RRNS. This allows RRNS to be efficiently applied to exascale systems improving their fault-tolerance and energy-efficiency.

2. Propose the RRNS API compatible with the current Habanero C/C++ library (HClib)[27], which demonstrates the feasibility of thread-level RRNS in the current Asynchronous

[1]Mean Time Between Failures (MTBF) $>$ Average Human Lifespan (100 years).

Many-Task (AMT)[28] programming model widely used for programming exascale systems.

3. Through further optimizations of thread-level RRNS microarchitecture, this method shows 62.25% and 58.67% reduction respectively in energy and Energy Delay Product (EDP), compared with the state-of-the-art Asynchronous Many-Task (AMT) black-box resiliency method.

1.3 Thesis Statement

Error-tolerant microarchitectures via Redundant Residue Number System can further extend the semiconductor technology roadmap, and alleviate reliability and power challenges of exascale computing.

1.4 Thesis Outline

Chapter 2, 3, 4, and 5 of this thesis discuss the scalable energy-efficient and error-tolerant CPU microarchitecture designs for millivolt switches.

Chapter 2 introduces the basic RRNS concepts, approaches, and optimizations that are necessary for the CPU microarchitecture design. A scalable RRNS microarchitecture framework that can be compatible with error correction and detection&restart is also proposed. The publications related to Chapter 1 and Chapter 2 are [7, 26, 29, 30, 31].

Chapter 3 discusses the RRNS checkpointing&restart mechanisms in detail. First introduced the static checkpointing&restart method, and then implemented two adaptive checkpointting&restart strategies based on the system's historical information. Finally, it shows experimental results and analysis based on the (4, 2)-RRNS configuration.

Chapter 4 introduces the correction-only RRNS system and discusses the range of reasonable error correction capabilities for the RRNS microarchitecture framework.

Chapter 5 demonstrates the hybrid RRNS system with the capability of both correction

and checkpointing&restart. Then find out the optimal or suboptimal design configuration in the (n, r)-RRNS setting space with different fault-tolerant capabilities. The publication related to Chapter 3, Chapter 4 and Chapter 5 is [26].

Chapter 6 introduces thread-level RRNS and its related microarchitecture. This design aims to alleviate the critical exascale computing challenges regarding high error rates and high energy consumption. Finally, through experimental simulation, it is found that thread-level RRNS has better Energy Delay Product (EDP) results than thread-level TMR (Triple Modular Redundancy). The publication related to Chapter 6 is waiting to submit.

Finally, Chapter 7 summarizes the previous discussions and proposes future work that is related to this topic.

CHAPTER 2

SCALABLE RRNS MICROARCHITECTURES FOR MILLIVOLT SWITCHES

This chapter first describes the error-tolerant capabilities of one configuration point: (4,2)-RRNS. The first digit '4' refers to the number of non-redundant moduli/residues, and the second digit '2' refers to the number of redundant moduli/residues. Then, discussing the relationship between the number of moduli/residues and the capability of fault-tolerance in the entire design space. This chapter also introduces several detailed considerations that are essential for the scalable RRNS microarchitecture design, including signed number representation, correction factors, arithmetic overflow detection, comparison, multiplier optimization, and RRNS base selection. Finally, a scalable RRNS microarchitecture that can support error correction and error detection&roolback is proposed for the millivolt switches.

2.1 (4,2)-RRNS Error Detection/Correction Algorithm

The algorithm for one error detection/correction was originally given by Watson [25]. However, RNS renders comparison and arithmetic overflow detection to be a non-trivial exercise. This section presents algorithms to perform these functions by augmenting the error checking algorithm. This way, no extra hardware is warranted beyond that required by the error checking.

The one error correction algorithm proposed by Watson [25] is based on an error correction table. The working steps of this algorithm for a system with 4 non-redundant moduli (m_1, m_2, m_3, m_4) and 2 redundant moduli (m_5, m_6), for any given integer X ($< M = m_1 m_2 m_3 m_4$) is as follows: (a) Use a base-extension Algorithm [32, 33, 25] to compute $|X'|_{m_5}$ and $|X'|_{m_6}$, where $|X'|_{m_5}$ and $|X'|_{m_6}$ are the outputs of the the base-extension algorithm algorithm. The inputs to the base-extension algorithm are from the outputs of

Table 2.1: Error Correction table of RRNS System with Moduli (3,5,2,7,11,13)

Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ	Δm_5,Δm_6	i' ϵ
1 , 10	4 6	4 , 5	1 1	5 , 12	3 1	7 , 7	4 3	9 , 3	2 2
2 , 10	2 3	4 , 6	4 4	6 , 1	3 1	7 , 8	1 2	10 , 3	4 1
2 , 12	4 6	4 , 7	2 1	6 , 4	2 4	8 , 1	2 2		
3 , 3	1 1	4 , 11	4 5	6 , 5	4 3	8 , 4	4 2		
3 , 9	4 5	5 , 8	4 4	7 , 2	4 2	8 , 10	1 2		
3 , 12	2 3	5 , 9	2 1	7 , 6	4 2	9 , 1	4 1		

non-redundant subcores: $|X|_{m_1}$, $|X|_{m_2}$, $|X|_{m_3}$ and $|X|_{m_4}$, where $|X|_m = X \ mod \ m$; the computational output of the subcore with m_i modulus. (b) For $i = 5, 6$: compute $\Delta m_i = ||X'|_{m_i} - |X|_{m_i}|_{m_i}$. (c) A non-zero difference indicates the presence of an error. This pair of differences may index into an entry of a pre-computed (fixed) error correction table, which contains the index of the residue that is in error and a correction offset that needs to be added to that residue to correct said error.

The RRNS Error detection/correction algorithm could easily be activated via scheduling the RRNS check instruction, which has been defined in CREEPY ISA [30]. For the error detection step, the system would perform (a) and (b) to get values of Δm_5 and Δm_6. For the error correction step (if necessary), it performs (c). Analysis of the algorithm reveals that the error detection step would take 8 cycles while the correction step takes 2 cycles. Therefore, once the system inserts an RRNS_check instruction, the first step is to execute the 8-cycle error detection procedure. If no error is found, then this RRNS_check instruction is complete and it takes 8 cycles in total. But if an error is detected, then we need 2 more cycles for the RRNS correction operation to complete (resulting in 10 cycles in total).

For ease of presentation, we present such an error correction table for a smaller (toy) set of RRNS base moduli in Table 2.1. The total entries in such a table is at most $2 \sum_{i=1}^{4} (m_i - 1)$. For the remainder of this chapter, this set of bases are used for explanatory purposes.

1 Error	Binary	%3	%5	%2	%7	%11	%13
Intended	168	0	3	0	0	3	12
Actual	42	0	2	0	0	3	12

$r'_5 = |42|_{11} = 9 \quad \Delta r_5 = |r_5 - r'_5|_{11} = 5$

$r'_6 = |42|_{13} = 3 \quad \Delta r_6 = |r_6 - r'_6|_{13} = 9$

2 Error	Binary	%3	%5	%2	%7	%11	%13
Intended	0	0	0	0	0	0	0
Actual	160	1	0	0	6	0	0

$r'_5 = |160|_{11} = 6 \quad \Delta r_5 = |r_5 - r'_5|_{11} = 5$

$r'_6 = |160|_{13} = 4 \quad \Delta r_6 = |r_6 - r'_6|_{13} = 9$

One Error Correction Lookup Table with moduli (3,5,2,7,11,13)

$\Delta r_5, \Delta r_6$	i', ϵ	$\Delta r_5, \Delta r_6$	i', ϵ	$\Delta r_5, \Delta r_6$	i', ϵ
1,10	4,6	4,11	4,5	7,7	4,3
2,10	2,3	5,8	4,4	7,8	1,2
2,12	4,6	5,9	2,1	8,1	2,2
3,3	1,1	5,12	3,1	8,4	4,2
3,9	4,5	6,1	3,1	8,10	1,2
3,12	2,3	6,4	2,4	9,1	4,1
4,5	1,1	6,5	4,3	9,3	2,2
4,6	4,4	7,2	4,2	10,3	4,1
4,7	2,1	7,6	2,4		

Figure 2.1: (4,2)-RRNS is capable of 1EC or 2ED

2.2 Error-tolerant Capability of (4,2)-RRNS

Similar to other resilient techniques, the RRNS error-tolerant capability depends on the amount of redundancy, i.e., the number of redundant moduli. For example, a *(4,2)-RRNS* is capable of *either* 1EC *or* 2ED, but not both simultaneously. The term '1EC' refers to '1 Error Correction', and '2ED' stands for "2 Error Detection". The reasoning for this, as well as the working of the RRNS detection and correction algorithms, are now explained by means of an example.

Consider there is a (4,2)-RRNS with toy bases (3,5,2,7,11,13). Say the result from an operation is intended to be (0,3,0,0,3,12) whereas, due to a transient fault, the second residue changes from 3 to 2 (Figure 2.1). If an RRNS error detection is performed immediately, the Chinese Remainder Theorem [34] or the Base Extension Algorithm [35, 25] can be used to re-generate the redundant residues ($r_5' = 9$, $r_6' = 3$) from the non-redundant residues ($r_1 = 0$, $r_2 = 2$, $r_3 = 0$, $r_4 = 0$). Now, the `delta_value_pair` ($\Delta r_5 = 5$, $\Delta r_6 = 9$) is used to infer the possibility of an error and potentially restore the correct result. Three cases arise for the `delta_value_pair`:

1. Both are zero. This implies that there are no errors in the RRNS data. No correction

or recovery is necessary.

2. 1EC mode:

   - Exactly one of them is non-zero. This indicates that the corresponding non-zero redundant residue is in error. Correction is performed by overwriting the errant redundant residue with the re-generated value.
   - Both are non-zero. It means that exactly one of the non-redundant residues is in error, and a single error correction LUT[1] must be consulted to handle the error.

3. 2ED mode: One of them is non-zero. This implies that at least one residue is in error. No LUT consultation is necessary for error detection, but an efficient checkpointing/restart mechanism is necessary (Chapter 3)

In 1EC mode, the `delta_value_pair` ($\Delta r_5 = 5$, $\Delta r_6 = 9$) should be used as an input to index into the one error correction LUT. This returns (i'=2,ϵ=1), which translates to the following: a correction offset of '1' is to be added to the 2^{nd} residue in order to correct its errant value. However, consider the example of the double errors in Figure 2.1 where two residues are actually incorrect. Observe that the `delta_value_pair` is again ($\Delta r_5 = 5$, $\Delta r_6 = 9$), and 1EC mode performs incorrect restoration by accessing LUT, resulting in system failure because its correction error model tolerates at most one residue in error. However, if the *(4,2)-RRNS* system was working in a 2ED mode rather than 1EC, then such a double error would indeed be detected. In other words, *(4,2)-RRNS* is capable of exactly one of 1EC or 2ED. If both 1EC and 2ED are simultaneously supported in a (4, 2)-RRNS system, it will cause similar `delta_value_pair` conflictions as shown in Figure 2.1, resulting in incorrect behaviors. More redundant information is needed to support both 1EC and 2ED in the same RRNS.

[1]The one error correction Lookup Table (LUT) is unique to the RRNS base set, and its size grows linearly with the magnitude of the base moduli (summation).

2.3 (n,r)-RRNS Error-Tolerant Extension

The number of non-redundant moduli (n) and redundant moduli (r) can be adjusted for a performance-energy-reliability tradeoff. n has a direct impact on the amount of bit-level parallelism that one can leverage. Of more fundamental interest to this thesis, r has a direct impact on error tolerance capabilities, as summarized in the two lemmas below by [36]:

1. An (n, r)-*RRNS* code can detect upto r residue errors, *or* correct upto $\lfloor \frac{r}{2} \rfloor$ reside errors.

2. An (n, r)-*RRNS* code can correct upto t residue errors and simultaneously detect upto v ($> t$) errors if $t + v \leqslant r$.

The precondition for these lemmas is that the redundant moduli must be larger than the non-redundant ones. This condition, along with other considerations for designing a set of RRNS base moduli for energy-efficient and error-resilient microarchitectures are described in detail in Section 2.9.

Using the lemmas, as an example, a *(4,3)-RRNS* architecture can either employ 0EC3ED or 1EC2ED, while a *(4,4)-RRNS* architecture can choose between 0EC4ED and 1EC3ED. The key advantage of a hybrid (correction+detection&rollback) architecture is that thanks to the correction, rollbacks become statistically less expensive in terms of latency and energy. In theory, timely correction can avoid the frequency of rollbacks at a certain probability. On the other hand, with the help of detection&rollback, a higher error (multiplicity) tolerant capability is achieved, which can in turn enable higher energy savings due to lowering V_{dd} further. These benefits are, of course, subject to the runtime overheads of error correction and detection&rollback, therefore resulting in an expansive RRNS design space. In other words, this thesis explores the optimal or suboptimal values for n, r, c and d in an *(n,r)-RRNS-cECdED* architecture, given the constraints above.

Increasing n improves the parallelism in RRNS operations, upto a certain point, after which the overheads of routing and duplication overshadow the benefits. Furthermore, too

large a value can hinder base selection in its ability to find co-primes, since larger the value of n, lower is the bit width of each modulus for a given range. Finally, a higher value of n also slows down the RRNS-Binary Conversion Unit (as shown in Figure 2.11). Hence, the reasonable value of n is limited to $\{3, 4, 5\}$ in RRNS design space exploration.

Increasing r improves reliability in general and can therefore help reduce V_{dd} and save energy. However, after a certain point, the added overhead of the redundant residues and the Error Detection Units (as shown in Figure 2.11) outweigh the savings due to lower V_{dd}. As a thumb rule, $r > n$ is avoided as the bit widths of redundant residues are at least as wide as the non-redundant ones, and $r > n$ would therefore add significant overhead.

Given the nature of increasing either n or r as outlined above, minimizing Energy Delay Product (EDP) is therefore a discrete convex optimization problem in both these dimensions. As such, it is sufficient to evaluate the ranges of $3 \leq n \leq 5$ and $1 \leq r \leq n$ as long as the minimum does not lie on the boundary of this surface. The results from the empirical evaluation in Chapter 5.2 confirm this and finds that the checkpoint/restart mechanism in isolation is generally superior to other approaches for non-memory-intensive workloads. But for memory-intensive ones, the correction-only approach is the best bet.

2.4 Signed Number Representation

This chapter discusses and compares three competing ways of representing signed numbers. Each presents its set of trade-offs, which will be discussed in detail below. M is the product of all the non-redundant moduli (M = m1*m2*m3*m4 in (4,2)-RRNS) and MR is the product of all the redundant moduli (MR = m5*m6 in (4,2)-RRNS).

1. **Complement M*MR Signed Representation**

   The M*MR complement signed representation is depicted by Figure 2.2. To provide a few examples, 0 is represented by 0, 1 is represented by 1, $\frac{M}{2} - 1$ is represented by $\frac{M}{2} - 1$, -1 is represented by $M * MR - 1$ and $-\frac{M}{2}$ is represented by $M * MR - \frac{M}{2}$.

This is similar to signed binary representation. However, representing numbers in this manner breaks known error correction algorithms[25].

Figure 2.2: Complement M*MR signed representation

2. **Complement M Signed Representation**

The M complement signed representation is depicted in Figure 2.3. This is similar to the M*MR complement representation, except that the wrap-around occurs at M as opposed to M*MR. This representation does not break error correction algorithms, provided that some correction factors (scaling and offset) are applied to the result of each arithmetic operation. However, further analysis indicates that these correction factors require knowledge of the signs of the operands, which are not trivial to determine like in binary. The RRNS sign determination is a time-consuming algorithm. Moreover, THE overflow detection of an arithmetic operation is unknown for this representation.

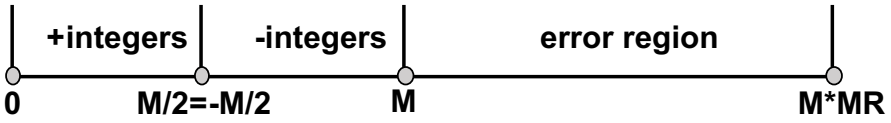


Figure 2.3: Complement M signed representation

3. **Excess-$\frac{M}{2}$ Signed Representation**

The Excess-$\frac{M}{2}$ signed representation is depicted in Figure 2.4. The excess notation, sometimes known as offset notation, merely shifts each number by $\frac{M}{2}$. To further elaborate, 0 is represented by $\frac{M}{2}$, 1 is represented by $\frac{M}{2}+1$ and -1 is represented by $\frac{M}{2}-1$. Similar to the M Complement representation, the results of arithmetic

operations must be offset by a correction factor before they can be corrected. However, these correction factors turn out to be independent of the sign of the operands. Moreover, this representation enables simple algorithms for comparison (and thereby sign detection) and overflow detection of an arithmetic operation. In fact, these algorithms make use of a technique used in the error detection/correction algorithm itself. These algorithms are discussed in detail in Section 2.6 and Section 2.7.

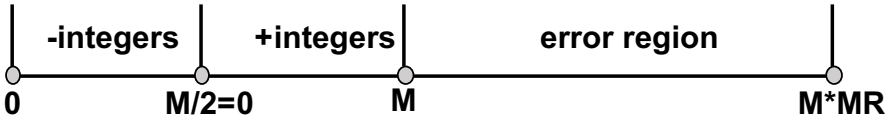


Figure 2.4: Excess -M/2 signed representation

Through the above comparisons, Excess-$\frac{M}{2}$ was chosen as the de facto signed representation scheme.

2.5 Correction Factors

This section discusses the addition, subtraction and multiplication operations on two numbers with correction factors that do not generate any overflow. Recall from Section 2.4 that suggests to use the Excess-$\frac{M}{2}$ notation, which means that there is a bijective mapping from any number x such that $-\frac{M}{2} < x < \frac{M}{2}$ to $x + \frac{M}{2}$. Because of this offset, arithmetic operations results need to be re-adjusted using values called *correction factors*. [However, this has nothing to do with the RRNS error correction operation.]

Addition Consider the addition of two numbers x and y. To represent the mapping, define a and b such that $0 \leq a, b < \frac{M}{2}$ so that there is no overflow.

Case 1: $x, y \geq 0$

Consider $x = a$ and $y = b$. The sum $x + y$ can be represented for each subcore $1 \leq i \leq n + r$ as follows:

$$\left| |\frac{M}{2} + a|_{m_i} + |\frac{M}{2} + b|_{m_i} \right|_{m_i} = \left| |M|_{m_i} + |a+b|_{m_i} \right|_{m_i} \tag{2.1a}$$

$$= |a+b|_{m_i} \; for \; 1 \leq i \leq n \tag{2.1b}$$

However, the expected addition result is:

$$\left| \frac{M}{2} + a + b \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |a+b|_{m_i} \right|_{m_i} \tag{2.2}$$

It follows that:

1. $1 \leq i \leq n$ and m_i is odd: Examining equations 2.1b and 2.2 imply that no correction factor is necessary.
2. $1 \leq i \leq n$ and m_i is even: Examining equations 2.1b and 2.2 implies that a constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.
3. $n+1 \leq i \leq n+r$: Examining equations 2.1a and 2.2 imply that a constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be subtracted from the result.

Case 2: $x, y < 0$

Setting $x = -a$ and $y = -b$, and re-working equations similar to Equations 2.1a, 2.1b and 2.2 result in correction factors that are identical to Case 1.

Case 3: $x > 0$, $y < 0$ (Without loss of generality.)

Setting $x = a$ and $y = -b$, and re-working equations similar to Equations 2.1a, 2.1b and 2.2 result in correction factors that are identical to Case 1.

Subtraction Due to the symmetric and offset based nature of the Excess-$\frac{M}{2}$ representation, just one of the working cases is presented; without loss of generality: $x = a$ and

$y = b$. Then, $x - y$ becomes:

$$\left| |\frac{M}{2} + a|_{m_i} - |\frac{M}{2} + b|_{m_i} \right|_{m_i} = |a - b|_{m_i} \tag{2.3}$$

However, the expected subtraction result is:

$$\left| \frac{M}{2} + a - b \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |a - b|_{m_i} \right|_{m_i} \tag{2.4}$$

From examining equations 2.3 and 2.4, it follows that:

1. $1 \leq i \leq n$ and m_i is odd: No correction factor is necessary.
2. $1 \leq i \leq n$ and m_i is even: A constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.
3. $n + 1 \leq i \leq n + r$: A constant correction factor of $|\frac{M}{2}|_{m_i}$ needs to be added to the result.

Multiplication Again, for brevity, we only present the case where two positive integers are multiplied; without loss of generality: $x = a$ and $y = b$; the product xy becomes:

$$\left| |\frac{M}{2} + a|_{m_i} |\frac{M}{2} + b|_{m_i} \right|_{m_i} = \left| |\frac{M^2}{4} + \frac{(a + b)M}{2}|_{m_i} + |ab|_{m_i} \right|_{m_i} \tag{2.5}$$

However, the expected multiplication result is:

$$\left| \frac{M}{2} + ab \right|_{m_i} = \left| |\frac{M}{2}|_{m_i} + |ab|_{m_i} \right|_{m_i} \tag{2.6}$$

As residues are typically 8-bit wide, consider a 511 entry LUT per subcore that stores the following:

$$LUT(s) = \left| \frac{M^2}{4} + \frac{(s - 1)(M)}{2} \right|_{m_i} \tag{2.7}$$

From examining equations 2.5, 2.6 and 2.7, it follows that:

1. $1 \leq i \leq n$ and m_i is odd: No correction factor is necessary.
2. $1 \leq i \leq n$ and m_i is even: The correction factor can be effected by computing $s = a + b$ and then subtracting $LUT(s)$ from the result of the multiplier.
3. $n+1 \leq i \leq n+r$: The correction factor can be effected by computing $s = a+b$ and then subtracting $LUT(s)$ from the result of the multiplier.

The addition and subtraction operations' correction factors require a single, constant addition/subtraction operation, whereas for multiplication, 2 additions/subtractions and a modest table lookup are required. Another advantage of the schemes presented here is that sign determination is not necessary and that they can be performed at the subcore level, without the involvement of the Residue Interaction Unit (RIU).

2.6 Overflow Detection

2.6.1 Unsigned Number Overflow Detection

In the absence of any error or overflow, adding 2 unsigned RRNS numbers results in both Δm_5 and Δm_6 being zero. As has been just explained, the presence of an error is handled by the error correction table. In the absence of error, it could be observed that any overflow manifests itself as a fixed index into the error correction table, with the entry not corresponding to any error. Table 2.2 provides some examples of this observation. While the computation of the deltas is most efficient using a base-extension algorithm, the Chinese Remainder Theorem(CRT) method is used to first convert the RRNS number to a binary number before computing the `delta_value_pair`. This is solely for explanatory purposes; binary conversion is not actually necessary to detect overflow.

By iterating through all possible combinations of numbers and operations, it can be observed that the `delta_value_pair` (Δm_5, Δm_6) of overflow is fixed. Moreover, (Δm_5, Δm_6) = (10,11) is not a legitimate address of the error correction table (Table 2.1), thus enabling a distinction between an error and an overflow. This approach, however, does

Table 2.2: Unsigned Number Overflow Examples in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	CRT/MRC	$\|X'\|m_5$,$\|X'\|m_6$	Δm_5, Δm_6
2+209	(2,2,0,2,2,2)	(2,4,1,6,0,1)	(1,1,1,1,2,3)	$(1,1,1,1) \Leftrightarrow 1$	$\|1\|_{11}$=1,$\|1\|_{13}$=1	10 11
3+209	(0,3,1,3,3,3)	(2,4,1,6,0,1)	(2,2,0,2,3,4)	$(2,2,0,2) \Leftrightarrow 2$	$\|2\|_{11}$=2,$\|2\|_{13}$=2	10 11
...	...	...	...	...	...	10 11
209+209	(2,4,1,6,0,1)	(2,4,1,6,0,1)	(1,3,0,5,0,2)	$(1,3,0,5) \Leftrightarrow 208$	$\|208\|_{11}$=10,$\|208\|_{13}$=0	10 11

Table 2.3: Excess-$\frac{M}{2}$ Overflow Examples for addition of two positive numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\|X'\|m_5$,$\|X'\|m_6$	Δm_5,Δm_6
1+104	(1,1,0,1,7,2)	(2,4,1,6,0,1)	(0,0,1,0,7,3)	(0,0,0,0,1,2)	$(0,0,0,0) \Leftrightarrow 0$	$\|0\|_{11}$=0,$\|0\|_{13}$=0	10 11
2+104	(2,2,1,2,8,3)	(2,4,1,6,0,1)	(1,1,0,1,8,4)	(1,1,1,1,2,3)	$(1,1,1,1) \Leftrightarrow 1$	$\|1\|_{11}$=1,$\|1\|_{13}$=1	10 11
...	...	...	...	...	...	...	10 11

not apply to multiplication.

2.6.2 Signed Number Overflow Detection

Recall from Section 2.4 that the Excess-$\frac{M}{2}$ signed representation is used. There are two independent scenarios of overflow:

1. *Two positive numbers addition* Table 2.3 provides a few examples illustrating the algorithm (Correction factors are explained in detail in Section 2.5). The $1 + 104$ in the first column is represented in decimal. After Excess-$\frac{M}{2}$ mapping, the computing equation is transformed to $106 + 209$ since $\frac{M}{2} = 105$ for the toy set of moduli. Therefore, the X RRNS value is the the RRNS of 106 and Y RRNS value is the RRNS of 209. We observe that the `delta_value_pair` (Δm_5,Δm_6) remains at a fixed value (10,11).

2. *Two negative numbers addition* Similarly, examples for adding two negative numbers are shown in Table 2.4. In this case, we can observe that the `delta_value_pair` (Δm_5,Δm_6) is fixed to (1,2).

Note that neither (10, 11) nor (1, 2) is a legitimate entry index of Table 2.1, thereby enabling a distinction between an error and an overflow. However, while this method works for both addition and subtraction, it does not hold for detection of multiplication overflow as the

Table 2.4: Excess-$\frac{M}{2}$ Overflow Examples for addition of two negative numbers in RRNS with Moduli (3,5,2,7,11,13)

X+Y	X RRNS	Y RRNS	X+Y RRNS	Add Correction Factors	CRT/MRC	$\lvert X'\rvert_{m_5}$,$\lvert X'\rvert_{m_6}$	Δm_5,Δm_6
-1-105	(2,4,0,6,5,0)	(0,0,0,0,0,0)	(2,4,0,6,5,0)	(2,4,1,6,10,12)	$(2,4,1,6) \Leftrightarrow 209$	$\lvert 209\rvert_{11}$=0,$\lvert 209\rvert_{13}$=1	1 2
-3-104	(0,2,0,4,3,11)	(1,1,1,1,1,1)	(1,3,1,5,4,12)	(1,3,0,5,9,11)	$(1,3,0,5) \Leftrightarrow 208$	$\lvert 208\rvert_{11}$=10,$\lvert 208\rvert_{13}$=0	1 2
...	...	...	...	...	...	...	1 2

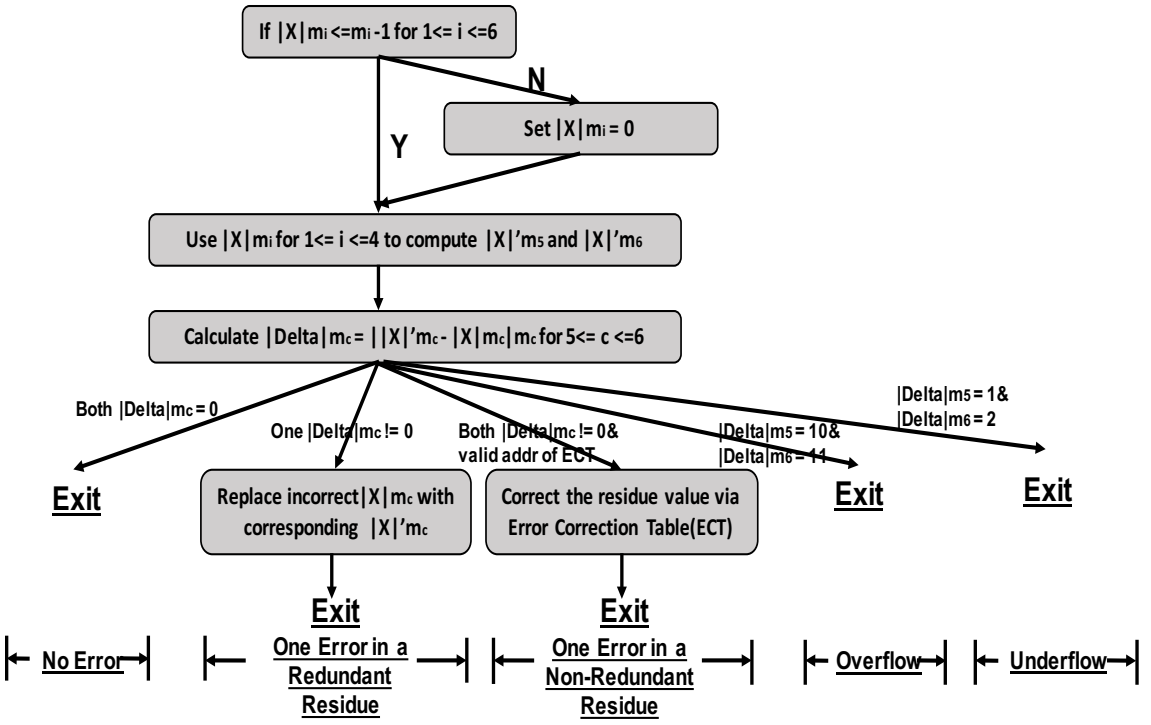


Figure 2.5: One error detection and correction algorithm with overflow/underflow detection

delta-pair is not constant and sometimes indexes into a legal error correction table entry.

Figure 2.5 shows the overview of One error detection and correction algorithm with overflow/underflow detection.

It can be observed that the described algorithm works in a similar manner even with the base sets that meet particular requirements. E.g. Waston's bases (199,233,194,239,251,509), an overflow results in a delta-pair of (77, 289), whereas an underflow results in (174, 220). Both these pairs do not index into legitimate entries of the error correction table for these set of bases (*cf.* Appendix E, Watson [25]).

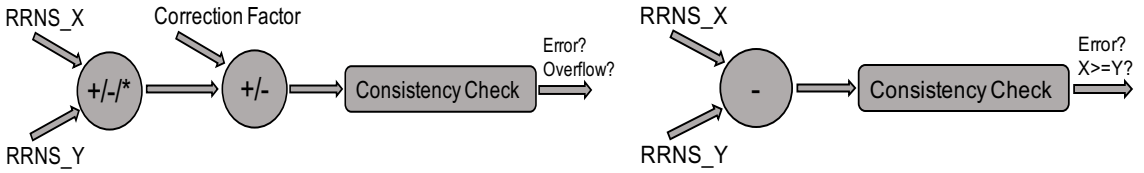
2.7 Comparison

Comparison is an essential operation because it may use in determining the direction of the control flow. In a manner similar to overflow detection, the potential algorithms are explored to perform RRNS comparison without incurring unnecessary hardware overhead.

Jen-shiun et al. [37] and Omondi [32] proposed number comparison methods for residue numbers based on parity bits. However, a prerequisite of these parity comparison methods is that all moduli are supposed to be odd (in addition to being pair-wise relatively prime). In some other systems, one of the non-redundant moduli is even (to enable fast fractional multiplication [25]), therefore this approach is not suitable in these scenarios.

Instead, this scheme leverages the error check algorithm itself to check for an overflow post a subtraction: to compare X and Y, perform $X-Y$ and derive the `delta_value_pair` (Δm_5, Δm_6). Then, $X \geq Y$ iff the `delta_value_pair` is (0, 0) (*i.e.,* no overflow) and $X < Y$ iff the `delta_value_pair` is (174, 220) (*i.e.,* $X - Y$ results in an underflow).

This new residue number comparison method can be used for both unsigned and Excess-$\frac{M}{2}$ signed numbers. It is straightforward that this idea is suitable for unsigned residue numbers: if $X < Y$, then $X - Y \notin [0, M)$, thereby resulting in an underflow. For an Excess-$\frac{M}{2}$ signed number X, an injective mapped residue number can be defined as follows: $X_{mapped} = \frac{M}{2} + X$. Therefore, $X \geq Y$ iff $X_{mapped} \geq Y_{mapped}$, which reduces to an unsigned comparison. A caveat to note is that correction factors should not be added for a comparison operation. These are summarized in Figures 2.6a and 2.6b. In summary, this RRNS comparison is performed via combining a simple subtraction with error checking. If a comparison instruction is followed by an error checking instruction, this comparison could be replaced with a subtraction. From another aspect, by executing this comparing algorithm, a free error checking can be achieved on the corresponding data.



(a) Signed Overflow Detection (b) Signed Comparison

Figure 2.6: Signed Overflow Detection and Comparison

2.8 Optimized Multiplier Unit Design

Many workloads in the domains of multimedia, image processing and digital signal processing are highly multiplication intensive [38] . Index-sum multiplication [39, 40] was proposed and its principle is analogous to using a *logarithm* operation, *i.e.,* a multiplication can be achieved via a table lookup, addition and a reverse table lookup, as summarized as follows for the product of two numbers X and Y:

1. Use a pre-defined mapping table to generate $index(X)$ and $index(Y)$.
2. Compute the sum $Z = index(X) + index(Y)$.
3. Use a pre-defined reverse mapping table to return the product XY as $reverse\_index(Z)$.

If the computing data is 32-bit, the sizes of mapping tables are huge and need high area and hardware overhead. RNS can significantly reduce the table size by separating the data bit-width [39, 40]. This idea is extended into RRNS by adjusting the RRNS bases (*cf. Section* 2.9) to be amenable to index-sum LUTs, the requirements for which, are summarized below.

Index-sum multiplication is based on the theory of Galois fields, which can be classified into 3 types: $GF(p)$, $GF(p^m)$ and $GF(2^m)$, where, p is an odd prime number and $m \in \mathbb{Z}^+$. The range of integers that can be represented bijectively in Galois fields, and the encoding methodology depends on the GF type[40]: (The methodology of deriving $GF(p^m)$ is skipped as these microarchitectures do not utilize this.)

Table 2.5: Mapping table of $GF(59)$ with a primitive root of 11 ($g = 11$)

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
α	0	7	2	14	42	9	10	21	4	49	1	16	25	17	44	28	48	11	34	56
X	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**	**40**
α	12	8	47	23	26	32	6	24	22	51	53	35	3	55	52	18	37	41	27	5
X	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**		
α	40	19	57	15	46	54	45	30	20	33	50	39	38	13	43	31	36	29		

Ex: $X = 3, Y = 6 \implies \alpha_X = 2, \alpha_Y = 9$, whose sum is 11, which reverse maps to 18 and is indeed the desired product.

$GF(p)$: Any integer $x \in [1, p-1]$ can be uniquely coded as a single integral index code α by the relationship $X = |g^{\alpha}|_p$, where $\alpha \in [0, p-2]$, and g is a primitive root such that $|g^{p-1}|_p = 1$. See Table 2.5 for an example.

$GF(2^m)$: Any integer $x \in [1, 2^m - 1]$ can be coded as a triple integral index code $< \alpha, \beta, \gamma >$ by the relationship $X = 2^{\alpha}|5^{\beta}(-1)^{\gamma}|_{2^m}$, where $\alpha \in [0, m-1]$, $\beta \in [0, 2^{m-2} - 1]$ and $\gamma \in [0, 1]$. See Table 2.6 for an example.

Therefore, the relative preference of GF types are $GF(p) > GF(p^m) > GF(2^m)$ as they require 1, 2 and 3 index codes respectively. Furthermore, a smaller value of p and m leads to a smaller LUT. These considerations impact the selection of RRNS base sets, as shown in Section 2.9.

Using the index-sum technique in conjunction with RRNS, the complexity of multiplication is able to be significantly simplified. Index-sum multiplication can be efficiently performed via a simple addition and two straightforward table lookup operations. This optimization achieves a reduction in ALU gate count using this approach by about 87% when compared to using a traditional multiplier in RRNS, which itself reduces the gate count by 52% when compared to a traditional non-error-correcting binary ALU, thereby realizing area, energy and reliability improvements, as demonstrated in Chapter 3.6.

Table 2.6: Mapping table of $GF(2^6)$

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
α, β, γ	0,0,0	1,0,0	0,3,1	2,0,0	0,1,0	1,3,1	0,10,1	3,0,0	0,6,0	1,1,0	0,5,1	2,3,1
X	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
α, β, γ	0,15,0	1,10,1	0,4,1	4,0,0	0,12,0	1,6,0	0,7,1	2,1,0	0,13,0	1,5,1	0,14,1	3,3,1
X	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
α, β, γ	0,2,0	1,15,0	0,9,1	2,10,1	0,11,0	1,4,1	0,8,1	5,0,0	0,8,0	1,12,0	0,11,1	2,6,0
X	**37**	**38**	**39**	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**
α, β, γ	0,9,0	1,7,1	0,2,1	3,1,0	0,14,0	1,13,0	0,13,1	2,5,1	0,7,0	1,14,1	0,12,1	4,3,1
X	**49**	**50**	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**	**60**
α, β, γ	0,4,0	1,2,0	0,15,1	2,15,0	0,5,0	1,9,1	0,6,1	3,10,1	0,10,0	1,11,0	0,1,1	2,4,1
X	**61**	**62**	**63**									
α, β, γ	0,3,0	1,8,1	0,0,1									

Ex: $X = 3, Y = 6$ map to, respectively, $< 0, 3, 1 >, < 1, 3, 1 >$, whose sum results in $< 1, 6, 2 >$. Since $\gamma \in [0, 1]$, the modulo sum results in $< 1, 6, 0 >$, which reverse maps to 18 and is the desired product.

2.9 Base Selection for Scalable Systems

Recall, RRNS bases (or moduli) play a vital role not only in determining the error detection/correction capability of the architecture, but also in determining the operating data range of an RRNS machine. Prior work by Waston [25] provides the necessary and sufficient conditions of RRNS moduli in 1EC and multiple error detection. Preethy et al. [39, 40] discuss the limitations of index-sum multiplication optimization. The moduli conditions of scalable architecture are summarized in Table 2.7. The size of the error correction table exhibits exponential growth if the correctable capability needs to be improved. Therefore, correcting two or more residues in error is not an efficient configuration. So in the RRNS scalable architecture design, the number of correctable residues in a correction-only or hybrid system is limited to 1. Based on the property of the target architecture, system designers just need to pick the necessary conditions from the table. E.g., for a 1ECxED hybrid system, the conditions for both single error correction (*) and multiple error detection (+) are needed. Watson [25] also proposes an efficient algorithm for RRNS fractional multiplication, but this benefit introduces a new condition ($|m_a m_b - m_c m_d|$ = 1). The conditions in Table 2.7 are enough to cover the moduli selection of scalable RRNS architectures.

The total number of *Core Bits* (the bit width of one RRNS base set), which is re-

Table 2.7: Conditions for choosing RRNS moduli:
Multiple error detection (+), single error correction (*), fractional multiplication optimization(#), index-sum multiplication optimization (O), number of redundant moduli (r), number of detectable errors (e).

	Conditions for choosing RRNS base moduli
*+	All the residues in base set must be co-prime.
+	$r \geqslant e$.
+	For $x \in [1, r]$, the product of any x+r-e redundant moduli should be larger than the product of any x non-redundant moduli
*	$r \geqslant 2$.
*	$max_{n+1 \leqslant i \leqslant n+r} \frac{M_R}{m_i} \geqslant max_{1 \leqslant j \leqslant n} m_j$; M_R refers to the product of all the redudant moduli.
*	$M_R \geqslant max_{1 \leqslant i \neq j \leqslant n} m_i m_j$; M_R refers to the product of all the redudant moduli.
*	$M_R \neq 2m_i m_j - n_1 m_i - n_2 m_j; 1 \leq i \neq j \leq n; 1 \leq n_1 \leq m_j - 1; 1 \leq n_2 \leq m_i - 1$
*	$M_R \geq 2\sum_{i=1}^{n}(m_i - 1) + \sum_{j=n+1}^{n+r}(m_j - 1)$
#	$\lvert m_a m_b - m_c m_d \rvert$ = 1
O	Each m_i must be one of: prime (p), a power of prime (p^m), or a power of 2 (2^m); The order of preference is p > p^m > 2^m

lated to area, hardware and energy overhead and *Range*, where a larger range is better, are key factors to be considered when choosing the bases. Table 2.8 lists the base set candidates of the *(4,2)-RRNS-DED* system. For a range close to a 32-bit binary representation, (139,349,128,379,503,509) is a good base set that meets all the requirements. A lower energy, albeit lower range alternative which still meets all the conditions could be (113,239,128,211,241,251). It should be noted that for a higher range than 32-bit binary (211,421,256,347,503,509) is a good candidate, which more than doubles the range, at an expense of 1 additional core bit.

Subcore Bits	Core Bits	Range	Possible Base Sets	Base Format
$7, 8, 7, 8, 8, 8$	46	467921792	(97,223,128,169,241,251)	$(p, p, 2^7, 13^2, p, p)$
$7, 8, 7, 8, 8, 8$	46	729405056	**(113,239,128,211,241,251)**	$(p, p, 2^7, p, p, p)$
$8, 8, 7, 8, 8, 8$	47	635871872	(151,167,128,197,241,251)	$(p, p, 2^7, p, p, p)$
$7, 8, 8, 7, 9, 9$	48	430002432	(89,233,256,81,503,509)	$(p, p, 2^8, 3^4, p, p)$
$7, 9, 7, 8, 9, 9$	49	951568256	(109,283,128,241,503,509)	$(p, p, 2^7, p, p, p)$
$7, 9, 7, 8, 9, 9$	49	1032240512	(89,361,128,251,503,509)	$(p, 19^2, 2^7, p, p, p)$
$8, 8, 8, 8, 8, 9$	49	2149852322	(199,233,194,239,251,509)	$Waston$
$9, 7, 7, 9, 9, 9$	50	1082179712	(491,67,128,257,503,509)	$(p, p, 2^7, p, p, p)$
$7, 9, 7, 9, 9, 9$	50	1406512512	(81,463,128,293,503,509)	$(3^4, p, 2^7, p, p, p)$
$7, 9, 8, 8, 9, 9$	50	1230080256	(81,433,256,137,503,509)	$(3^4, p, 2^8, p, p, p)$
$8, 9, 7, 9, 9, 9$	51	2353365632	**(139,349,128,379,503,509)**	$(p, p, 2^7, p, p, p)$
$8, 9, 8, 9, 9, 9$	52	7891035392	**(211,421,256,347,503,509)**	$(p, p, 2^8, p, p, p)$
$9, 9, 8, 9, 9, 9$	53	7710332672	(277,317,256,343,503,509)	$(p, p, 2^8, p^m, p, p)$

Table 2.8: Possible *(4,2)-RRNS-DED* Base Sets

2.10 RRNS Fixed Point Arithmetic

The IEEE 754 floating-point arithmetic works efficiently for the binary number system thanks to its efficient bit-shift operations (for normalize/denormalize). However, for RRNS, although bit-shifting operations are able to be implemented as multiplications and optimized scaling algorithms [41, 33], these need a large amount of overhead and make it very inefficient. Furthermore, these algorithms involve minimal residue arithmetic and require the centralized algorithm processed in RIU, thereby not really benefitting from RRNS. As to overcome these issues, this section presents two alternatives of RRNS fixed point arithmetic which can replace the traditional IEEE floating point for RRNS based microarchitectures.

2.10.1 2-RRNS Concat Representation

2-RRNS Concat proposes using two RRNS integer values to represent the integer and fractional segment of a fixed-point number separately. RRNS fixed-point addition and multiplication can be easily performed by using this scheme. In addition, the carry bit from the lower fractional segment to the higher integer segment is computed via overflow detection,

which can also be merged with an RRNS consistency check [30]. Hence, this addition requires two (or three) RRNS integer additions and one overflow detection.

Multiplication is illustrated in Figure 2.7 and Figure 2.8. In order to maintain bit-width consistency, the highest (overflow) and lowest (negligibility) segments of the result are cut in this method. Steps 1-4 compute the intermediate results. In Step 1, fractional multiplication [25] is used to compute $\lfloor \frac{XY}{M} \rfloor$. Step 2 involves a normal RRNS multiplication between the integer parts of the two inputs, and the products across the fractional/integer portions are computed via Step 3 and Step 4. Overflows are detected via the fractional multiplication algorithm on the inputs of Step 3 and Step 4. Hence, this multiplication methodology requires three RRNS integer multiplications, three RRNS fractional multiplications, six RRNS additions, and two overflow detections.

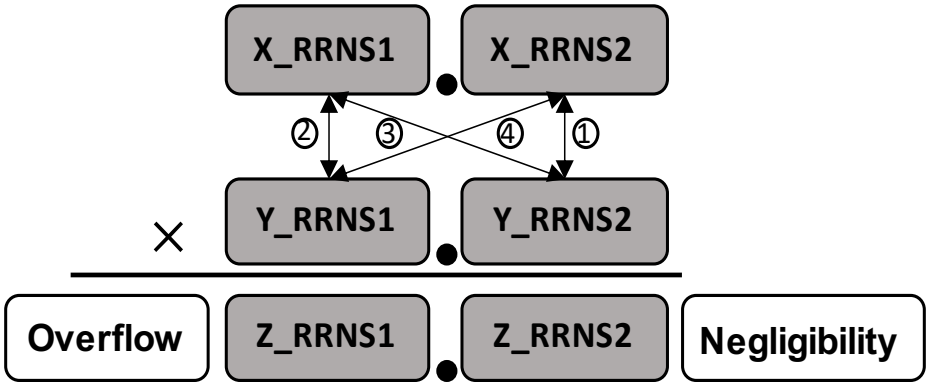


Figure 2.7: 2-RRNS Concat Multiplication

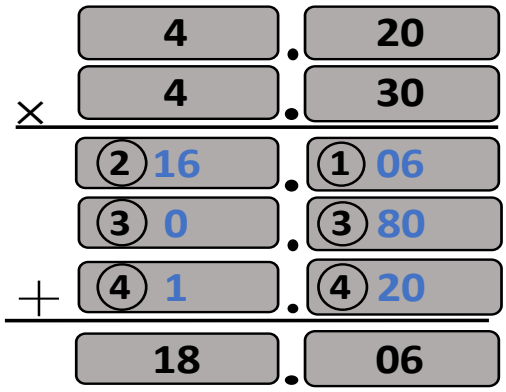


Figure 2.8: An Example of 2-RRNS Concat Multiplication

2.10.2 RRNS Logical Partition Representation

RRNS Logical Partition's key idea is to use a single RRNS integer number to represent a fixed point number for both integer and fractional parts. In this section, the example's values are represented by decimal and binary for simplicity. Assuming that the value range of an RRNS is 2^n, then the corresponding binary system range is n-bit. For an n-bit binary computation input, logically, the higher $\frac{n}{2}$ bits are the integer segment and the remaining lower bits make up the fractional part. Figure 2.9 shows examples of how to convert the conventional fixed-point values to RRNS Logical Partition representation. If using this presentation, no extra work is needed to take care of carry bit of this fixed point addition. Hence, addition is identical to a standard RRNS integer addition.

In order to keep the computation result inbound, the lowest $\frac{n}{2}$ bits of the output of the fixed-point multiplication should be discarded. To meet this requirement, a preprocessing step of the multiplication inputs is necessary. Thus, this multiplication includes the following three steps: 1) The preprocessing step right-shifts $\frac{n}{4}$ bits (or divide by a constant in RRNS) for both input X and input Y. 2) a regular RRNS integer multiplication 3) The final output is the fixed point product. The overflow could be detected via a fractional multiplication algorithm if the system needs to support overflow detection. Hence, the minimum requirement of this multiplication is two RRNS scaling down and one RRNS integer multiplication. Figure 2.10 presents an example of Logical Partition Multiplication, and Figure 2.11 shows how to convert integer representation output back to fixed-point representation. This conversion back operation is unnecessary during the execution. The system can directly use the Logical Partition representation as input or output.

The Logical Partition representation has a significantly lower overhead for multiplication compared to the 2-RRNS Concat representation from the previous discussion. However, performance comes at the cost of loss in precision and range. The fractional number representation selection depends on the target application, and the tradeoff between performance and range/precision should be taken into account.

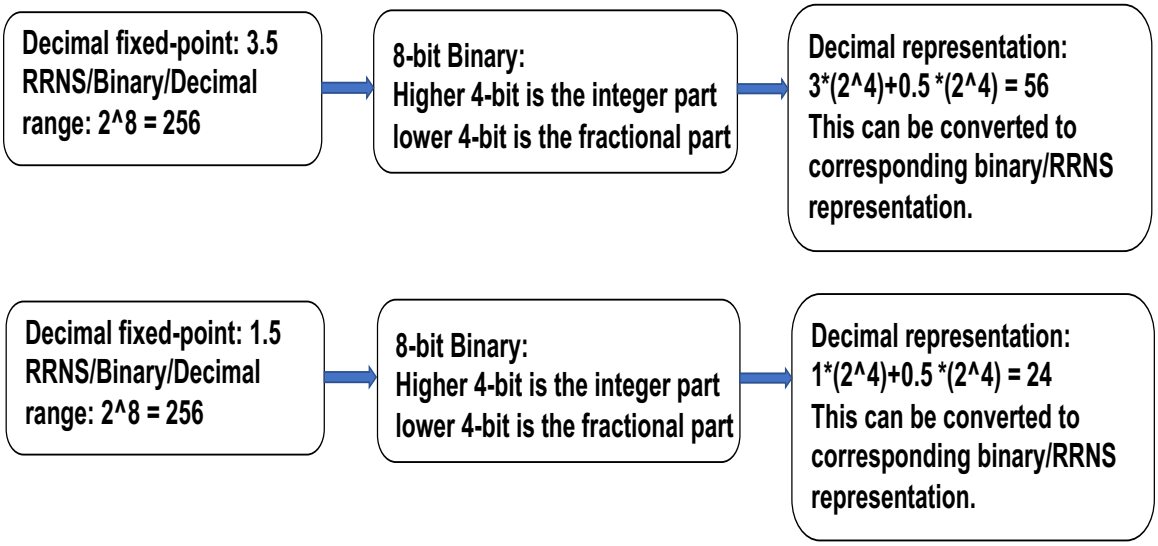


Figure 2.9: Converting fixed-point representation to integer representation

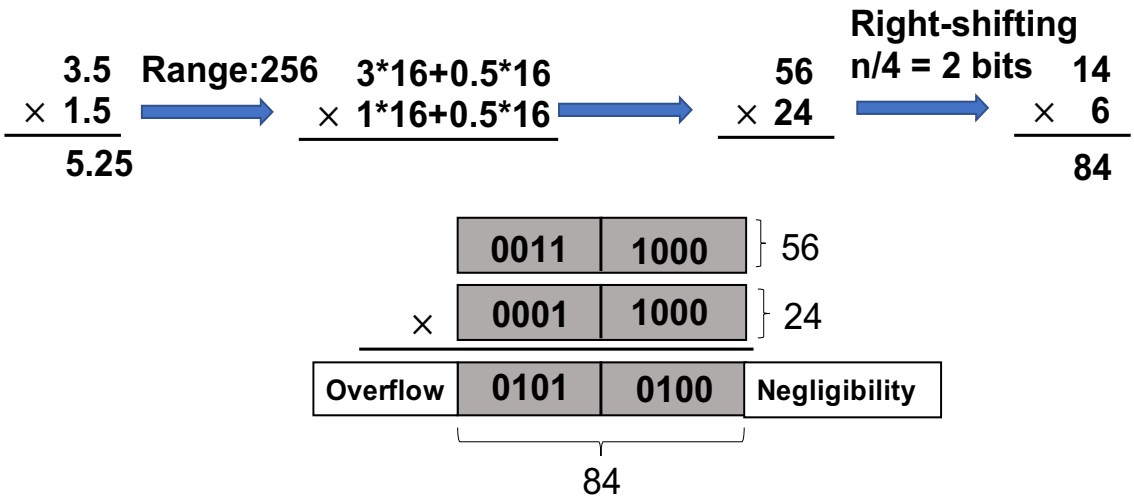


Figure 2.10: Logical Partition Multiplication

Figure 2.11: Converting integer representation back to fixed-point representation

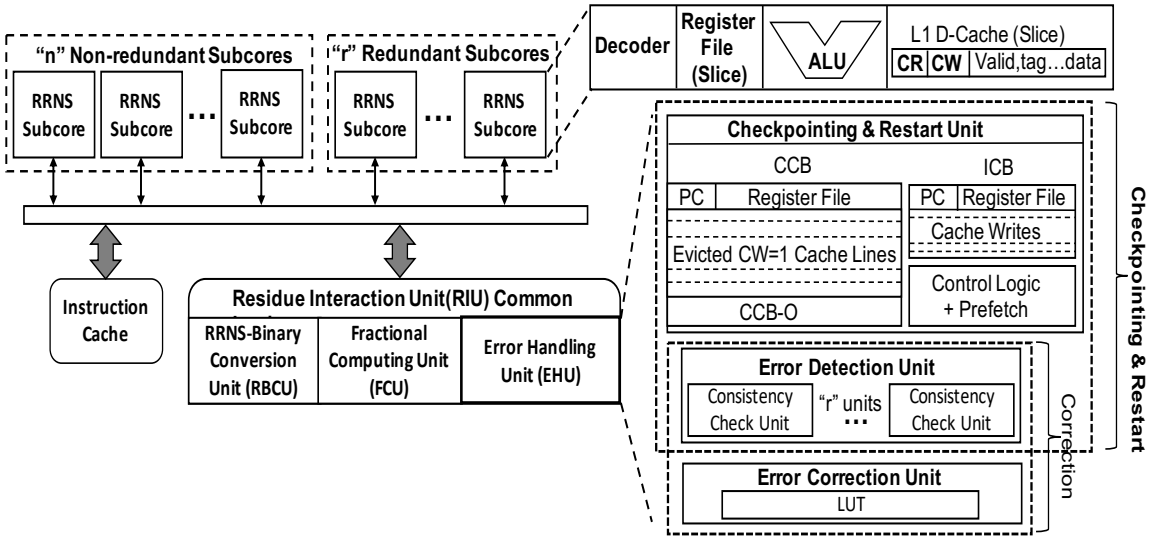


Figure 2.12: The overview of a scalable *(n,r)-RRNS* microarchitecture capable of checkpointing/restart and error correction

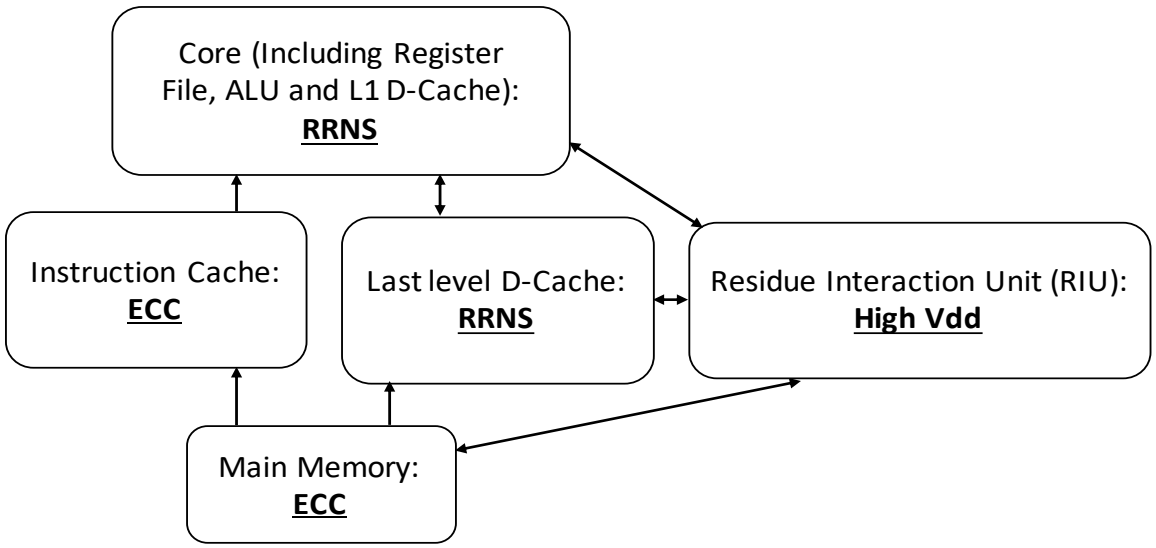


Figure 2.13: Error-tolerant techniques of components

2.11 A Scalable (n,r)-RRNS Microarchitecture Supporting Detection and Correction

Figure 2.12 shows the microarchitecture of a general *(n,r)-RRNS* core. Each RRNS core consists of $n + r$ subcores, where n subcores are non-redundant, and r subcores are redundant. For example, in the case of *(4,2)-RRNS-2CR*, there are 4 non-redundant subcores and 2 redundant subcores. The instruction cache and instruction register are ECC protected

and are shared by all subcores. The D-Cache is distributed among the subcores such that each subcore stores only the slice that corresponds to its RRNS residue. A subcore also contains its own ALU with addition/subtraction and multiplication, and its slice of the register file. Further, all the subcores share the Residue Interaction Unit (RIU), which houses the RRNS-Binary Conversion Unit (RBCU), the Fractional Computing Unit (FCU) and the Error Handling Unit (EHU). This scalable microarchitecture in this subsection is extended from the single error correction microarchitecture (CREEPY [30, 29]). Different error-tolerant capabilities can be achieved via adjusting the number of RRNS subcores and consistency check units. Extra hardware resources corresponding to checkpointing&restart mechanism have also been included. Other basic designs such as instruction set architecture do not need to be changed.

Figure Figure 2.13 shows the error-tolerant techniques of different system components. Both main memory and instruction cache use ECC technology, while high Vdd is applied in RIU to ensure error-free of corresponding RRNS algorithms. RRNS protects all the remaining units. Considering the locality and reducing RRNS-binary conversion overhead, register file and data caches are in RRNS format.

RRNS-Binary Conversion Unit (RBCU) serves as a data format converter, which is necessary for RRNS division and bit-shifting operations. Fractional Computing Unit (FCU) is used to handle the RRNS fractional number computation. The Error Handling Unit (EHU) contains the Checkpoiting&Restart Unit, the Error Detection Unit and the Error Correction Unit. The Error Detection Unit is necessary for both correction and checkpointing schemes. Finally, the main memory is protected via ECC, and therefore is assumed to be error-free in this work.

While the correction scheme requires a LUT to correct errors, the checkpointing scheme tracks changes between Complete Checkpoint (CC) intervals using modified data cache blocks and two additional buffers. The Complete Checkpoint (CC) refers to a copy of the entire system state updates. Unlike Incremental Checkpoint (IC) [42], CC can be applied

as inputs such that the system is able to *directly* rollback to a previously valid state. The IC update has to go through the copies one by one in sequential order. Loads and stores between two CCs are tracked using a combination of the cache and a checkpoint buffer (Complete Checkpoint Buffer (CCB) in Figure 6.4). Each cache block contains an extra bit, 'CR', which is set when the block is read from. The existing dirty bit of the writeback cache is re-used as a 'CW' bit, to mark blocks written to. Cache evictions are stored in the CCB as (address, value) pairs for fast, energy-efficient lookup during checkpoint destruction (described later in Section 3.2). If the CCB buffer has a conflict or gets full, an overflow buffer (CCB-O) of significantly smaller capacity is available for storing these records.

The Consistency Check Unit responsible for generating the `delta` values from Section 2.2 resides in the Error Detection Unit, and is shared by the Checkpoiting&Restart Unit and the Error Correction Unit. However, since its design-specifics are orthogonal to the main focus of this work, it has not been explicitly shown in Figure 6.4. In general, a scheme that has r redundant cores requires r Consistency Check Units. This presents an energy-reliability trade-off based on the number of errors the architecture chooses to correct or restart. For example, *(4,1)-RRNS-1CR*, while using only one Consistency Check Unit, saves dynamic computation and consistency check energy, will not be able to detect 2-errors. Therefore, its Mean Time Between Failures (MTBF) would be lower for the same signal energy than that of a *(4,2)-RRNS-2CR* since the latter will be able to catch all 1-residue and 2-residue errors (and some multiple-errors).

2.12 Related Work

2.12.1 RNS and RRNS

The energy efficient properties of RNS due to its low-bit-width operations and absence of carries across residues have found applications in the digital signal processing (DSP) [14, 15, 16] domain. Furthermore, the representability of high bit-width integers as a tuple-of-

resides has been leveraged by the cryptography (RSA) [17, 18, 19] community. Anderson [22] proposed an architecture and ISA for an RNS co-processor designed to run datapath operations in tandem with a general-purpose processor running binary instructions, where the primary role of the general purpose processor is to handle control flow. The RNS co-processor uses an accumulator-based ALU and does not support caching or computational error correction (RRNS). Furthermore, it requires a conversion to binary (and vice-versa) for comparison operations, which is expensive. Clearly, this architecture is able to be further improved. A unique feature of their ISA is their ability to encode instructions targeting two ALUs simultaneously. But this can easily be extended to architectures in this thesis and enable such superscalar-like capabilities if need be.

Chiang et al. [37] provide RNS algorithms for comparison and overflow detection, but assume all bases to be odd and do not consider error correction. Similarly, Preethy et al. [39, 40] integrate index-sum multiplication into RNS, but do not consider its impact on the properties of RRNS bases critical to this chapter.

Ever since Watson and Hastings [33, 25, 35] introduced RRNS as an efficient means for computational error correction, there has been a significant body of research [43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 34, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69] that strives to improve upon it. These are orthogonal to this chapter, and further such algorithmic research can be used to optimize aspects of the core itself, such as the RIU.

2.12.2 Other Error-tolerant Techniques

RAZOR[70] and DECOR [71] are timing speculation methods which make corrections via circuit timing violations. However, these methods use classical transistors as the system infrastructure. If exploring the potential energy benefit by decreasing Vdd, the frequency must reduce and result in setup time violations. DIVA[72] achieves redundancy by partially duplicating the pipeline. DIVA essentially belongs to Double Modular Redundancy while maintaining the simple core without errors. That is to say, the overhead to keep the

computational reliability must be more than 100%.

2.12.3 State-of-the-art Checkpointing Schemes

Checkpointing is a widely used resilient methodology in exascale systems, especially when the error rate soars as the design complexity increases. Thus, the checkpointing becomes an indispensable resilient candidate of exascale systems[73]. Plank et al.[74] proposed diskless checkpointing to eliminate the performance degradation causing by remote disk accesses. However, diskless checkpointing has to allocate extra memory space for the correct checkpoint copy. Because this copy is usually huge, diskless checkpointing may pollute the memory and reduce its use efficiency.

Young [75] and Daly [76] presented mathematical models to compute optimal **static** Long Intervals (LI) from the first-order to the higher-order estimation. However, they lack Incremental Checkpointing (IC) considerations and assume that recovery time is the same throughout execution.

Chinchilla [77] is a recent proposed adaptive checkpointing scheme. Its basic idea is to deploy as many checkpoints as necessary initially to ensure continuous execution. It then removes the unnecessary checkpoints by the history status. However, Chinchilla has the following limitations: 1) The potential market of Chinchilla is for the intermittent computing of energy-harvesting devices. In this system, a checkpoint should be inserted before devices run out of energy. After the energy is restored, the system uses the last checkpoint to reboot the program and make sure the application makes progress. That said, the check frequency of this adaptive mechanism is limited by the energy availability of the devices. 2) Chinchilla does not incorporate error detection. The goal of their mechanism is to ensure continuous execution after energy restoration. 3) They use basic blocks as checkpoint intervals. However, the size of the basic block is usually small (4-6 instructions). Thus, the check frequency is much higher than necessary if used in an RRNS energy-efficient system. Using Chinchilla would eliminate the energy savings getting from lowering Vdd.

Levitin et al. [78] designed another dynamic checkpointing policy for heterogeneous standby systems. However, its limitations are summarized below: 1) They assumed the error detection is perfect. Their detection model can report an error immediately when it happens, and the time for detection is zero. In other words, they did not consider the system overheads of monitoring errors. 2) Their dynamic method needs to know the total operations (M) of the entire task. Extra efforts to compute M are required. 3) We find that incremental checkpointing may significantly reduce the re-execution overhead. However, they did not consider this optimization. 4) Moreover, this mechanism is optimized for a standby system that needs N elements to achieve reliability. The RRNS checkpointing tolerant technique discussed in this thesis needs only a single element.

Amnesic Checkpointing and Recovery(ACR) [79] is orthogonal methodology. It reduces the overhead by recomputing a subset of checkpoint entries. However, ACR deploys modular redundancy to detect computational errors. From the discussion in Section 1.1, modular redundancy has been summarized as an inefficient method. Furthermore, their checkpoints are uniformly distributed over the execution. That said, their checkpointing intervals are static. We improve this by proposing two new adaptive schemes.

Thus, new RRNS checkpointing mechanisms are necessary to further exploring the potential of energy reduction. The adaptive checkpointing methods proposed in this thesis improve feasibility and efficiency by overcoming problems from the state-of-the-art. Moreover, checkpointing mechanisms in this thesis constrain the current dirty machine states in cache hierarchy and overflow buffers, providing the possibility of response time reduction in recovery and update.

CHAPTER 3

(N,R)-RRNS DETECTION&RESTART-ONLY SYSTEMS FOR MILLIVOLT SWITCHES

A core that leverages error detection alone, such as *(4,2)-RRNS-2ED*, requires an efficient rollback mechanism to avoid error diffusion. In this section, first the overview of the checkpointing&rollback scheme is described, followed by the details of checkpointing operations (Chapter 3.2), the hardware overhead (Chapter 3.3) and the conditions for moduli selection (Chapter 2.9). Finally, two adaptive interval adjustment schemes (SOE and EIH in Chapter 3.4) are proposed to improve energy and EDP further.

3.1 RRNS Checkpointing&Restart Overview

Traditional checkpointing mechanisms maintain an extra checkpointing copy in memory. This memory state copy includes Program Counter (PC), register file as well as all the identical memory footprints from the beginning of the application. Maintaining such a complete copy therefore reduces memory capacity. Even when an efficient compression technique [80, 81] is applied, each process still must sacrifice a large amount of extra main memory capacity to record its verified checkpoint copy/copies. The approach in this chapter avoids this pitfall by adding an extra 'CR' bit in each cache line and using *Long Interval* (LI) based *Complete Checkpoints* (CC) in conjunction with *Short Interval* (SI) based *Incremental Checkpoints* (IC).

The active CC is maintained using the *Complete Checkpoints Buffer* (CCB) (Section 3.2) and the cache hierarchy itself (via the CR/CW bits in cache lines). As shown in Figure 3.1, a CC starts at the beginning of each LI-cycle period. The current CC is automatically updated by the store operations until the end of this LI. At the same time, the LI is further divided into several SIs where ICs are inserted to reduce re-execution overhead.

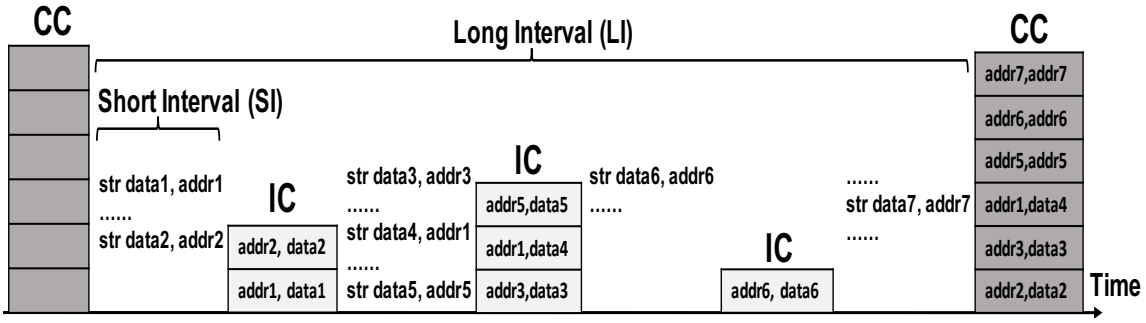


Figure 3.1: RRNS Checkpoint Overview

Whenever a modified line is written to the CCB/CCB-O (eviction victim), it is verified by the Consistency Check Units, in parallel with program execution. At the end of the LI, the RF and the modified lines still present in the cache hierarchy are also verified. Note that read lines do not need to be verified as incorrect reads can be detected as they spread to the RF or become visible via subsequent writes that use these reads, both of which would be checked at the end of an LI. This optimization is possible only because RRNS does not require a consistency check after each operation (unlike residue checking). The 'CR' bit is still necessary to track the lines that were actually read and to thereby avoid a deadlock, which would occur if the erroneous read is not remedied (by using a clean copy from main memory) upon rollback.

As depicted in Figure 3.2, at the end of the LI (*Long Interval*), if no error is detected during verification, the modified lines are written back to the main memory and a new CC (*Complete Checkpoints*) is created. However, at any point, if an error is detected, the system rolls back to the latest error-free state and execution restarts from that point with the help of the ICs (*Incremental Checkpoints*), which are applied in chronological order. At the time of restoring from an IC, it is checked for errors in a procedure similar to the CC. More details of checkpoint ***creation***, ***verification***, and either ***rollback*** or ***commit*** are now presented in Section 3.2.

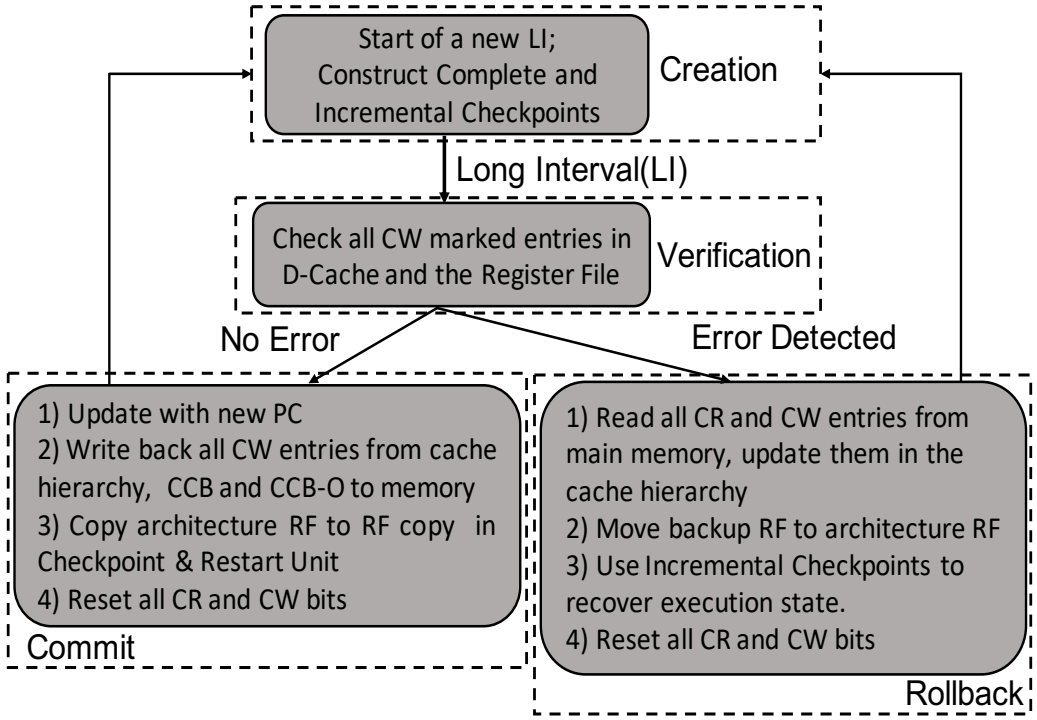


Figure 3.2: RRNS Checkpointing&Restart Flowchart

3.2 RRNS Checkpointing Operations

At the beginning of *Complete Checkpoints* (CC) ***creation***, the current (verified) register file (RF) is copied into the *Complete Checkpoints Buffer* (CCB)'s RF, along with the current Program Counter (PC). All load and store instructions executed during the current *Long Interval* (LI) mark the 'CR' and 'CW' bits of the D-Cache blocks respectively. Cache evictions during this time, which have the 'CW' bit set, are captured in the CCB in the common-case (or the CCB-O if CCB has no space). It should be noted that RRNS checks are performed on these evicted lines (in a pipelined fashion) before they are stored in the aforementioned buffers. If an error is found at this time, an immediate rollback is initiated. As they are contained within the RIU, the CCB and CCB-O operate at high Vdd, and are therefore error-resilient.

During *Incremental Checkpoint* (IC) ***creation***, performed every *Short Interval* (SI) cycles, the PC and RF snapshots, along with any stores, are saved into the *Incremental Check-*

point Buffer (ICB). The ICB is in essence a streaming (next-line-prefetch) buffer, and is periodically flushed to a reserved segment of main memory called the *Incremental Checkpoint Segment* (ICS). The role of ICS is to facilitate ***rollback*** from an IC in the future.

The ***verification*** of the CC involves a cache sweep where the (pipelined) RRNS error detection is performed for all lines in the cache hierarchy with the 'CW' bit set high, and for all the registers in the architecture register file. This ***verification*** starts right after the last cycle of current LI. When the state of the machine is found to be error-free during verification, the checkpoint ***commit*** operation is performed. All the cache lines with the 'CW' bits set high are written to the CCB (or CCB-O), and all 'CW' and 'CR' bits are reset to 0. The next checkpoint is then created, and execution resumes. The CCB and CCB-O are written back to memory while the core continues execution. The CCB and CCB-O act like conventional store buffers, in that they have the ability to provide data to the cache on a miss. Moreover, the Incremental Checkpoints are marked invalid in the ICS at this time so that newer incremental checkpoints can overwrite them, ensuring that the size of the ICS remains sufficiently small.

Conversely, if the machine state is found to have at least one error or a cache line eviction was found to contain an error, the ***rollback*** procedure is triggered. During rollback, the RF and PC are restored to the copies that were created during checkpoint ***creation***. All cache lines with the 'CW' and 'CR' bits set are read from main memory, and the ICs are applied serially (oldest to most recent) to the machine state to reduce re-execution of instructions. The ICs are streamed back from their segment in memory (ICS) into the ICB (using the simple next-line-prefetch logic to hide memory latency). The ICB is initially filled while the D-Cache is being swept for 'CW' and 'CR' entries. Once the error detection for an IC begins, the remaining entries for that IC and the next are streamed from the ICS. After an IC is completely verified (as valid), the PC, RF and cache are updated.

Since the ICB is logically split into two (with separate read and write ports), machine state recovery by the current IC and the next IC error detection can happen in parallel. This

process continues until there are no more valid ICs left or an IC is found to contain an error. At this point the 'CR' and 'CW' bits are reset, and execution resumes. While this rollback changes the start point of LI, note that a subsequent error will still rollback to the previous CC since no new complete checkpoint is created at this re-execution point. It should also be noted that re-execution will have a variable cache hit rate since the cache state does not perfectly rollback to its pre-execution state. This does not impact correctness in any manner.

3.3 RRNS Checkpointing Hardware Overheads

This section captures the hardware overheads of RRNS checkpointing schemes. In most cases, an 8-way 32KB CCB (*Complete Checkpoints Buffer*) and an 8-way 1 KB CCB-O (Overflow) are adequate for buffering dirty lines evicted from the cache. If an overflow of CCB-O itself is detected, the system halts the current LI (*Long Interval*) and moves on to checkpoint ***verification*** and potentially starts a new LI. These buffers can also be co-located with existing lower-level caches via set or way partitioning. The sizes of these buffers can potentially be further reduced by utilizing a segment of main memory and smartly prefetching the required data. For the ICs (*Incremental Checkpoints*), the ICB (*Incremental Checkpoint Buffer*) is used as a streaming buffer to hide DRAM (*Incremental Checkpoint Segment* (ICS)) access latency. It is assumed that the ICS is accessible every cycle, with a head fill latency of 100 cycles. As such, 1 KB is sufficient for the ICB. After a successful verification of a given IC i, it proceeds to the phase of state recovery by IC i. In order to continue verification of IC $i + 1$ in parallel with that, the ICB capacity needs to be doubled. Thus, the ICB is designed as two 1 KB FIFOs.

3.4 Adaptive Checkpointing Schemes

Having described the overall checkpointing mechanism and microarchitecture, the natural question that arises is regarding the frequency of CC and IC (Section 3.1). That is to say,

the intervals, LI (*Long Interval*) and SI (*Short Interval*) may significantly affect workload execution characteristics such as reliability, the number of rollbacks and re-play overhead. Young [75] and Daly [76] propose theoretical methodologies to compute optimal **static** Long Intervals (LI) from the first-order to the higher-order estimation. However, they lack IC (SI) considerations, and assume that the recovery time is the same throughout execution.

The best static configuration is highly dependant on the application. Furthermore, the best static configuration (on average) is 42% better (EDP) compared to the worst static configuration. Clearly, there is a need to design intelligent adaptive schemes, two of which are proposed in Sections 3.4.1 and 3.4.2.

3.4.1 Stochastic Overhead Estimation (SOE)

As evidenced by prior work above, LI is critical to optimizing the trade-off in performance, energy and reliability. A coarse-grained self-adjusting model is now presented that estimates the next LI based on history information, assuming a fixed SI of 5000 cycles in this example.

Upon a successful Complete Checkpoint (CC) commit, Formulae (1-3) of overhead estimation are computed:

$$Inv\_Exe + Sum\_CCs + Sum\_ICs \tag{3.1}$$

$$Inv\_Exe\_D + Sum\_CCs \times 0.5 + Sum\_ICs \tag{3.2}$$

$$Inv\_Exe\_H + Sum\_CCs \times 2 + Sum\_ICs \tag{3.3}$$

Formula (3.1) computes the total overhead due to checkpointing between the last 2 errors detected; Formula (3.2) estimates the total overhead of doubling LI in that duration; Formula (3.3) estimates the total overhead of halving LI in that duration. The terms of formulas are explained in Table 3.1.

Table 3.1: Equation Terminologies

Term	Explanation
Sum_CCs	Total overhead due to CCs' creation and verification between the last 2 errors
Sum_ICs	Total overhead due to ICs' creation and verification between the last 2 errors
Inv_Exe	The invalid execution overhead between the last 2 errors. An example of the invalid execution is shown in Figure 3.3. The further computation is available in Equation (3.4)
Inv_Exe_D	The estimated invalid execution overhead between the last 2 errors if the LI value is doubled
Inv_Exe_H	The estimated invalid execution overhead between the last 2 errors if the LI value is halved
LI	Time interval between 2 CCs
ave_LI	Average value of LI between the last 2 errors
$E(X)$	Expected value of cycle when error was generated in the last LI
num_ICs	Number of ICs in the last LI

With the checkpointing recovery mechanism's help, the machine state could roll back to the last valid IC or CC. For the purposes of model-based, stochastic estimation, assume the worst-case execution scenario wherein no immediate rollback upon error detection during the course of LI is done. In other words, any error is detected only at the end of the LI. Then, the execution between the last error-free IC (or CC) and the last cycle of the current LI is invalid. The execution in this period is called ***Inv_Exe*** and is shown in Figure 3.3. Formula (3.4) approximates the value of *Inv_Exe*. Once calculating the expected error encountering cycle (i.e., E(X), which will be discussed later), the number of Short Intervals (SIs) after the error encountering cycle can be estimated by $\lfloor(1 - \frac{E(X)}{LI}) \times (num\_ICs + 1)\rfloor$. The number of total SIs in a LI is equal to ***num_ICs+1***. The SI that imports the error should be considered as a part of invalid execution because of its irrecoverability. Thus, '1' is added at the end of the numerator in Formula (3.4). Finally, the ***Inv_Exe*** is computed by the percentage of invalid execution in last LI multiplies the average Long Interval of the last two errors. Similarly, ***Inv_Exe_D*** refers to the estimation of the invalid execution overhead

if the LI value is double and ***Inv_Exe_H*** if the LI value is halved. Because the LI is double or halved in Formula (3.5) and Formula (3.6), the number of SI (i.e., num_ICs+1) and the estimation LI (i.e., ave_LI) should be double or halved respectively to meet requirements.

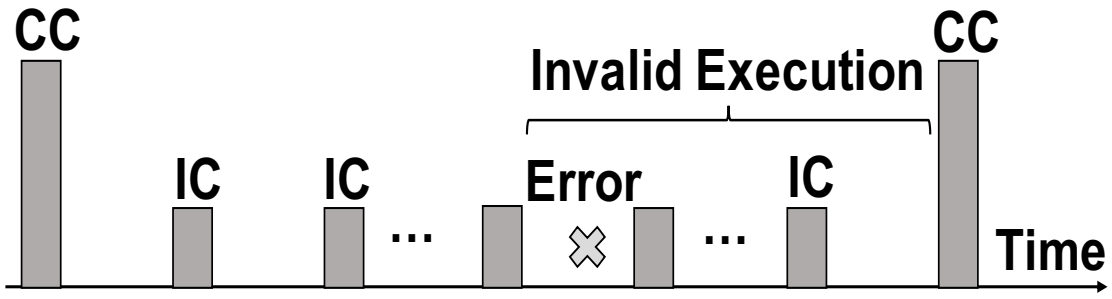


Figure 3.3: Invalid Execution Segment

$$Inv\_Exe = \frac{\lfloor (1 - \frac{E(X)}{LI}) \times (num\_ICs + 1) \rfloor + 1}{num\_ICs + 1} \times ave\_LI \quad (3.4)$$

$$Inv\_Exe\_D = \frac{\lfloor (1 - \frac{E(X)}{LI}) \times 2 \times (num\_ICs + 1) \rfloor + 1}{(num\_ICs + 1) \times 2} \times 2 \times ave\_LI \quad (3.5)$$

$$Inv\_Exe\_H = \frac{\lfloor (1 - \frac{E(X)}{LI}) \times 0.5 \times (num\_ICs + 1) \rfloor + 1}{(num\_ICs + 1) \times 0.5} \times 0.5 \times ave\_LI \quad (3.6)$$

The term ***overhead*** in this context can refer to latency (cycles) or energy. However, since accurate energy overheads are not easily obtained in general, latency is used instead. If Formula (3.1) returns the minimum value, then the LI value remains unchanged. If Formula (3.2) returns the minimum value, LI is doubled. Else, LI is halved. In other words, the goal of this adaptive model is to derive LI's value such that it minimizes the overhead of checkpointing.

The key parameter upon which the effectiveness of the model rests is **E(X)**: the expected value of the cycle at which the first error was generated in the last LI. It can be defined and derived as follows:

$$E(X) = \lim_{N\to\infty} \frac{\textit{Sum of Error Cycles in N Experiments}}{\textit{Num of Errors in N Experiments}} \tag{3.7}$$

$$= \lim_{N\to\infty} \frac{N * (1 * P_1 + 2 * P_2 + 3 * P_3... + LI * P_{LI})}{N * (P_1 + P_2 + P_3... + P_{LI})} \tag{3.8}$$

$$= \frac{(1 * P_1 + 2 * P_2 + 3 * P_3... + LI * P_{LI})}{(P_1 + P_2 + P_3... + P_{LI})} \tag{3.9}$$

, where, P_i in Formula (3.8) represents the probability of an error detected in the i^{th} cycle (given no errors in cycles upto $i-1$) of the last LI. If A is the probability of at least one transistor being in error in a specific cycle, then $P_i = A(1-A)^{i-1}$. Since A is instruction dependent, evaluating Formula (3.9) becomes intractable at runtime. A simplifying assumption is therefore made wherein each instruction is treated as a simple load from the cache such that A is a constant; $A = 1 - (1 - P_{err\_sram\_t})^{TCount}$, where $P_{err\_sram\_t}$ is the probability of an SRAM transistor experiencing a transient fault and $TCount$ is the active number of SRAM transistors relevant to the load operation.

As a result, Formula (3.9) can be re-written as:

$$E(X) = \frac{\sum_{i=1}^{L} i * A(1-A)^{i-1}}{\sum_{i=1}^{L} A(1-A)^{i-1}} \tag{3.10}$$

For the denominator of Formula (3.10):

$$\sum_{i=1}^{L} A(1-A)^{i-1} = \sum_{i=1}^{L} (1-A)^{i-1} - \sum_{i=1}^{L} (1-A)^{i} \tag{3.11}$$

$$= \frac{1-(1-A)^L}{1-(1-A)} - \frac{(1-A)(1-(1-A)^L)}{1-(1-A)} \tag{3.12}$$

$$= \frac{(1-A)^{L+1} - (1-A)^L + A}{A} \tag{3.13}$$

Similarly, for the numerator of Formula (3.10):

$$\sum_{i=1}^{L} i * A(1-A)^{i-1} = \sum_{i=1}^{L} i * (1-A)^{i-1} - \sum_{i=1}^{L} i * (1-A)^{i} \tag{3.14}$$

Define a variable T:

$$T = \sum_{i=1}^{L} i * (1-A)^i = (1-A) + \cdots + L(1-A)^L \tag{3.15}$$

Multiply both sides of Formula (3.15) by $(1-A)$:

$$(1-A)T = (1-A)^2 + 2(1-A)^3 + \cdots + L(1-A)^{L+1} \tag{3.16}$$

Subtract Formula (3.16) from Formula (3.15):

$$T - (1-A)T = (1-A) + \cdots + (1-A)^L - L(1-A)^{L+1} \tag{3.17}$$

$$= \frac{1-A-(1-A)^{L+1}}{A} - L(1-A)^{L+1} \tag{3.18}$$

From Formula (3.18), T can be computed:

$$T = \frac{1}{1-(1-A)} * (\frac{1-A-(1-A)^(L+1)}{A} - L(1-A)^{L+1} \tag{3.19}$$

$$= \frac{1-A-(1-A)^{L+1}}{A^2} - \frac{L(1-A)^{L+1}}{A} \tag{3.20}$$

$$\therefore \textit{Formula (3.14)} = (\frac{1}{1-A} - 1) * T \tag{3.21}$$

$$= \frac{1-(1-A)^L}{A} - L(1-A)^L \tag{3.22}$$

Substituting Formula (3.13) and (3.22) into Formula (3.10), and E(X) can be effectively computed as:

$$E(X) = Formula(3.10)$$

$$= [\frac{1-(1-A)^L}{A} - L(1-A)^L] \times \frac{A}{(1-A)^{L+1} - (1-A)^L + A}$$

$$= \frac{1-(1-A)^L - AL(1-A)^L}{(1-A)^{L+1} - (1-A)^L + A} = \frac{1-(1+AL)(1-A)^L}{A - A(1-A)^L} \tag{3.23}$$

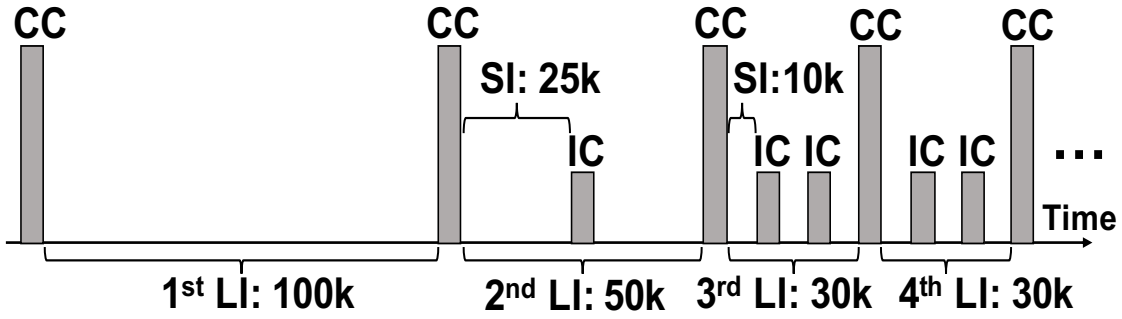


Figure 3.4: Error Interval Heuristics (EIH) Mechanism Examples; Default parameters: latest Error Interval(EI,200k cycles), minimal LI(30k cycles), minimal SI (10K cycles); No error detected

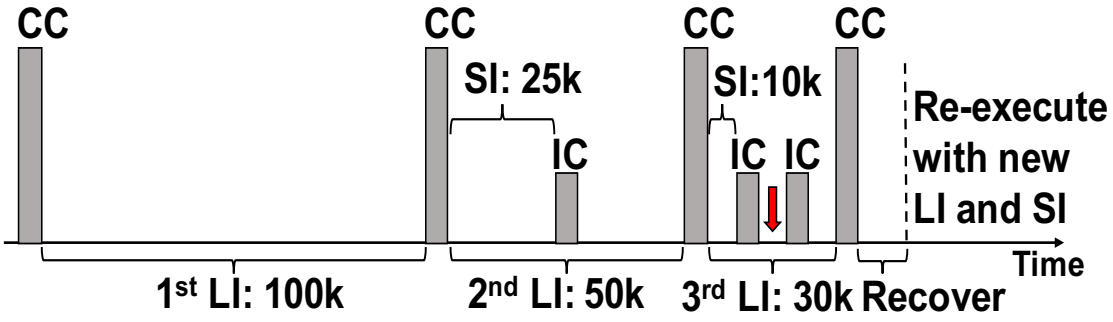


Figure 3.5: Error Interval Heuristics (EIH) Mechanism Examples; Default parameters: latest Error Interval(EI,200k cycles), minimal LI(30k cycles), minimal SI (10K cycles); An error is detected

3.4.2 Error Interval Heuristics (EIH)

The second adaptive checkpointing mechanism proposed is Error Interval Heuristics (EIH). EIH is designed based on the following assumption: since errors accumulate over time, the probability that an error is generated in the first half of an expected Error Interval (EI) is generally lower. EI is defined as the time interval between two consecutive errors. A higher LI with no IC is initially advisable, after which the checkpointing frequency may be gradually increased in an exponential manner. Note that reducing redundant checkpoints not only directly reduces latency and energy overheads, but also improves reliability by reducing the amount of time available for errors to creep in.

Figure 3.4 depicts the EIH mechanism before an error is detected. In this example, the latest EI is initialized to 200k cycles and the 1st LI is set to half the latest EI (100k) with no IC. Next, the system halves the old LI and inserts exactly one IC. Subsequently, the LI is halved and the number of incremental checkpoints is doubled, upto predefined lower bounds, such that the minimal LI and SI are 30k and 10k cycles, respectively. Once the bounds are reached, the intervals are kept unchanged until an error is detected.

As depicted in Figure 3.5, once an error is detected (marked with red arrow), either upon checking a cache eviction or at the end of the LI, a rollback is initiated (Section 3.1 and 3.2). Assuming the error is caught during the CC **verification** stage (at the end of the 3rd LI) in this example, the error detected cycle is $T = (100k + 50k + 30k +$ *CC_verification_overheads*$)$. Then the latest EI is re-initialized to T, and the process above is repeated until the workload is complete.

3.5 Evaluation Methodology

Evaluation of scalable RRNS cores should consider the aspects of reliability, performance, and energy. For the results discussed in this chapter, an in-order, unpipelined, trace-based cycle-accurate simulator was used [29, 30]. The simulator supports the proposed static and adaptive checkpointing schemes. For memory accesses, explicit conversion of RRNS based addresses to binary is not necessary. The RRNS format address may affect the data locality benefit and then reduce the system performance. Srikanth et al. [31] proposed and compared different methodologies to overcome this problem. Traces were generated using the trace-based debugging mode of gem5 [82]. The workloads used for evaluations are picked from SPEC2006, Mantevo and user-defined matrix multiplications. With the purpose of achieving a more thorough analysis on the scalable RRNS error-tolerant system, the workloads are divided into two parts: memory-intensive and non-memory-intensive. Gobmk, bzip2, mcf and miniFE are treated as memory-intensive while the rest are non-memory-intentive.

With the purpose of meeting the specific error model requirements of an RRNS architecture, a stochastic fault injection mechanism was designed and implemented in the simulation infrastructure. The error model and energy model of this section are the same as CREEPY [30, 29].The energy model for simulations is based on simple signal energy and active transistor count metric at a per operation granularity. Because these new millivolt switches do not have the leakage current problem, each instruction's energy consumption is counted through its activate transistor number and the corresponding signal energy value. Finally, the total energy consumption is calculated by accumulating the energy of each instruction in this workload. The signal energy input of the RRNS error-tolerant system is a 3-tuple with separated voltage domains [83, 84] for the general units (e.g., ALU), cache hierarchy (and RF), and RIU. Moreover, it should be noted that reduced V_{dd} transistors can still operate at frequencies similar to high V_{dd} transistors [85], allowing for a direct energy and delay comparison with a high V_{dd} binary core.

3.6 Experimental Results

Experimental results in this section are collected through models and methodologies described in Section 3.5. Section 3.6.1 shows the lowest signal energy values of workloads that meet the reliability threshold requirement. Other subsections use these lowest signal energy values as one of their inputs and explore the various configurations from different dimensions to evaluate system energy, delay, and Energy Delay Product (EDP).

3.6.1 Exploration of The Minimum Signal Energy

The minimal signal energy of common logic in *(4,2)-RRNS-1EC* systems ranges from 28-31 kT [30]. In theory, the signal energy of common logic could be further reduced by adopting *(4,2)-RRNS-2CR*, enabling stronger error detection capability. Before exploring the RRNS design space, two *(4,2)-RRNS* configurations are used as examples to compare thoroughly, and the energy reduction potential of the RRNS checkpointing&restart mech-

anism is evaluated. In order to measure the effect of signal energy, three metrics are presented: MTBF, energy, and EDP. For the trials in this subsection, LI and SI are fixed to 100k and 5k cycles respectively.

Figure 3.6 shows the MTBF of signal energies between 16-20 kT of *(4,2)-RRNS-2CR*, where an acceptable MTBF is greater than or equal to average human lifespan (100 years). From this figure, the minimal signal energy satisfying the MTBF constraint is 17 kT to meet the reliability requirements of all the benchmarks. Figure 3.7 illustrates the normalized energy consumption of signal energies between 17-23 kT, where energy consumption is normalized to 23 kT. For signal energies less than 17 kT, energy overhead becomes inhibiting due to the rapidly growing error rates. Energy is minimized for signal energies between 17-19 kT, depending on the workload. Similarly, Figure 3.8 indicates that an optimal EDP is achieved for signal energies of 17-19 kT. That is to say, the checkpointing system can further cut down the signal energy of common logic to close the Landauer limit. It should be noted that for all the following evaluations in this chapter, *(4,2)-RRNS-2CR* configurations are operated between 17-19 kT for different considerations.

As this is significantly lower than the state-of-the-art *(4,2)-RRNS-1EC* system that requires 28-31 kT, a corresponding EDP improvement in non-memory-intensive workloads is seen, which is depicted in Figure 3.9.

3.6.2 The Potential of Checkpointing&Restart Systems

In Figure 3.9, both checkpointing-only configurations, *(4,2)-RRNS-2CR* and *(4,1)-RRNS-1CR*, are static checkpointing results, which are obtained by setting the fixed LI = 100k cycles and the fixed SI = 5k cycles. All the values in this diagram are normalized to *(4,2)-RRNS-1EC*. Normalizing to *(4,2)-RRNS-1EC* allows for a direct comparison against related work [30, 35]. Note that with *(4,1)-RRNS*, only *1CR* meets the lemmas' requirements in Section 2.3.

Typically the checkpointing-only mechanism is not an appropriate candidate for memory-

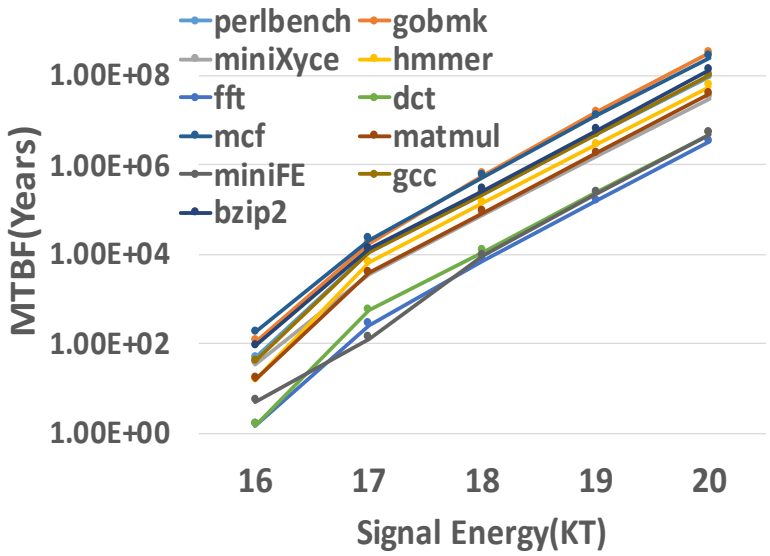


Figure 3.6: Minimal MTBF of computational logic in (4,2)-RRNS-2ED

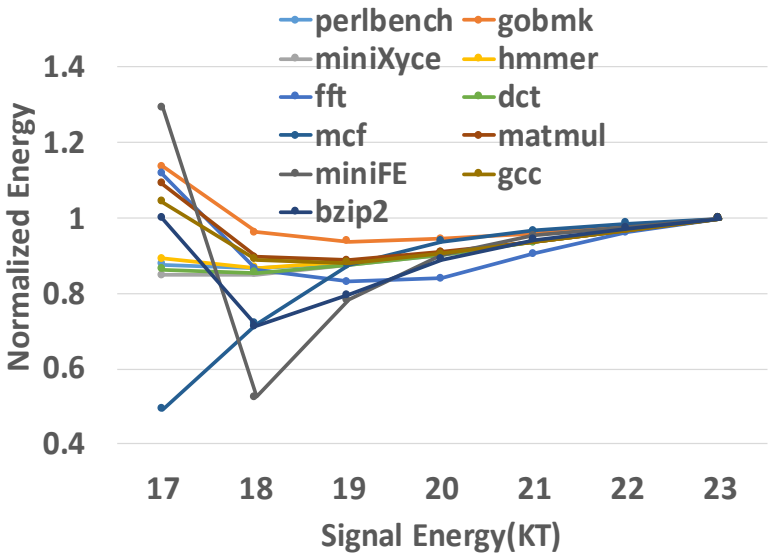


Figure 3.7: Minimal Energy of computational logic in (4,2)-RRNS-2ED

intensive applications because of the large-sized checkpoint creation and verification. In terms of EDP for memory-intensive jobs, the relatively better checkpointing-only scheme, *(4,1)-RRNS-1CR*, is 65% worse than *(4,2)-RRNS-1EC* on average. *(4,1)-RRNS-1CR* is intuitively considered as a low-cost candidate as it has a lower number of redundant residues. For the average non-memory-intensive results, *(4,2)-RRNS-2CR* beats the *(4,2)-RRNS-1EC* by 27% EDP reduction due to it manages relative lightweight checkpoint copies. This re-

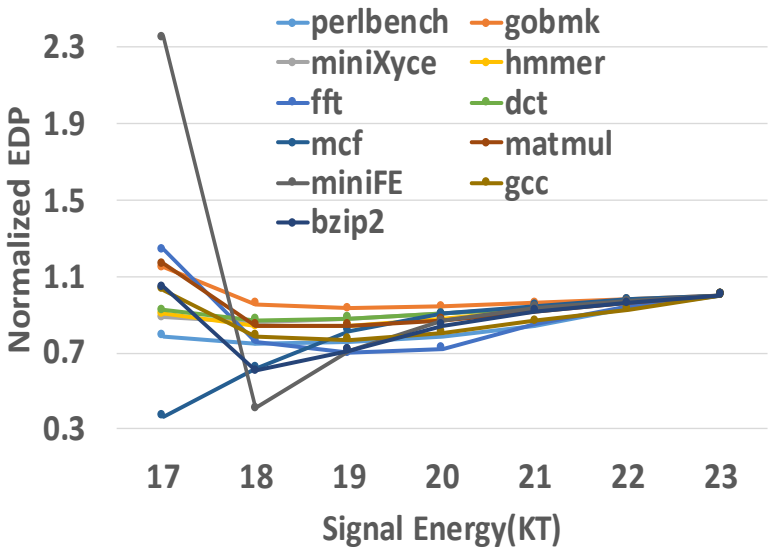


Figure 3.8: Minimal EDP of computational logic in (4,2)-RRNS-2ED

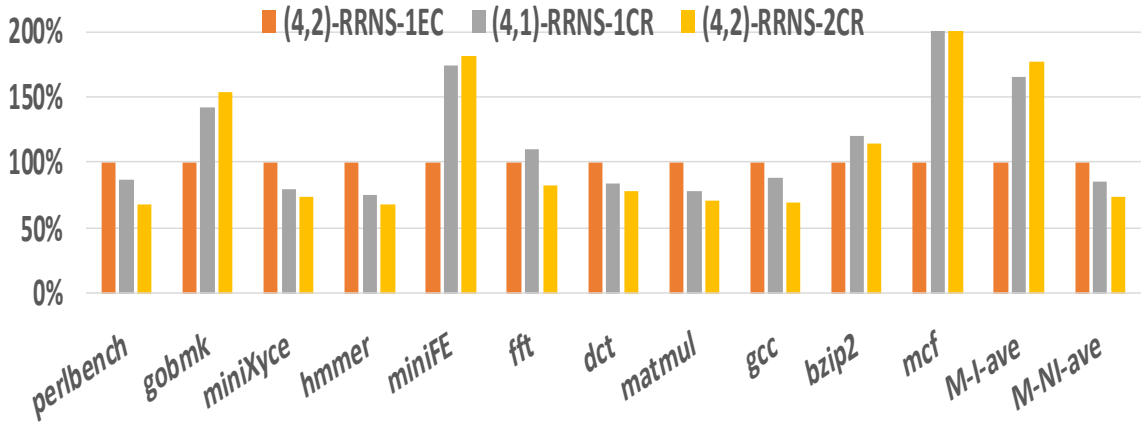


Figure 3.9: EDP of 1EC vs 1ED vs 2ED

sult confirms the EDP reduction potential of checkpointing&restart configurations. *(4,1)-RRNS-1CR* is 17% worse than *(4,2)-RRNS-2CR* on average, despite having lower subcore area and fewer Consistency Checking Units in the RIU. This is because of *(4,1)-RRNS-1CR*'s higher signal energy (23-25kT) requirement to satisfy the MTBF lifespan constraint. For fairness, we minimize signal energy for each configuration such that MTBF $\geqslant$ 100 years when comparing EDP in Figure 3.9 (23-25 kT, 28-31 kT, and 17-19 kT for *(4,1)-RRNS-1CR*, *(4,2)-RRNS-1EC*, and *(4,2)-RRNS-2CR* respectively). Further improvements can be drawn from profiled static configurations (static-best) and the adaptive schemes pro-

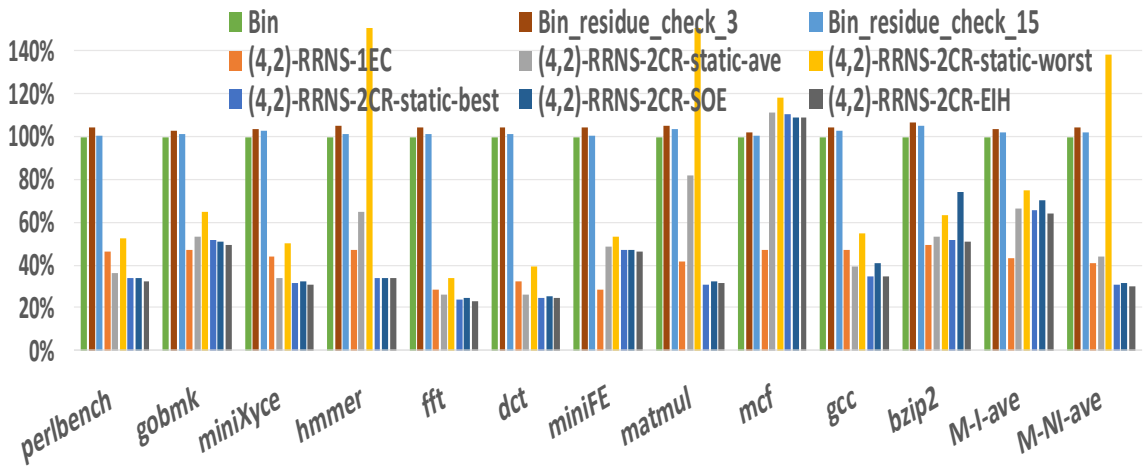


Figure 3.10: Comparison of Checkponting&Restart Energy

posed in Sections 3.4.1 and 3.4.2, as described next.

3.6.3 The Best Checkpointing & Restart Scheme

Figure 3.10 systematically compares the energy consumption of all the proposed checkpointing&restart schemes on *(4,2)-RRNS* configurations, normalized to a conventional binary core (*Bin*). *Bin_residue_check_x* (*x* refers to the modulus value) is the residue checking method discussed in Section 1.1. *(4,2)-RRNS-2CR-static-ave* denotes the arithmetic mean of 35 static configurations, which are obtained by varying LI and SI in the ranges 1k-500k and 1k-100k cycles respectively, such that SI $\leq$ LI. The three most notable RRNS checkpointing&restart schemes, ordered from least to most efficient, are *SOE*, *static-best*, and *EIH*. Here, *static-best* chooses, for each workload, the fixed SI-LI configuration that leads to its lowest EDP. On average, *EIH* realizes an energy reduction of 70%, 27%, and 1% over Binary, *(4,2)-RRNS-1EC*, and *(4,2)-RRNS-2CR-static-best* respectively for non-memory-intensive workloads.

The performance of RRNS configurations is always worse than *Bin*, as shown in Figure 3.11. This is because of the inherently slow comparison, bit-shift and division operations in the RNS domain. However, the performance overhead introduced by RRNS (to maintain reliability while also lowering system energy) can be made relatively insignifi-

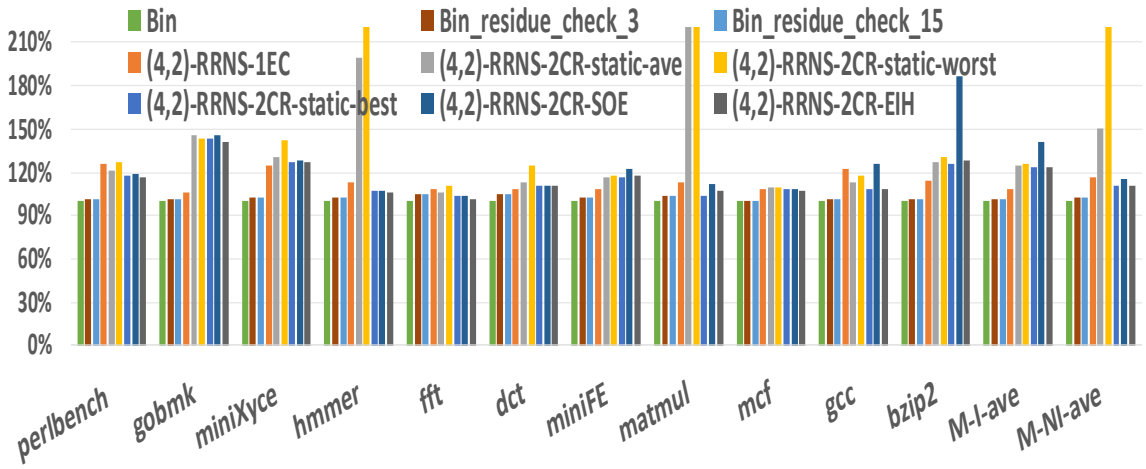


Figure 3.11: Comparison of Checkponting&Restart Runtime

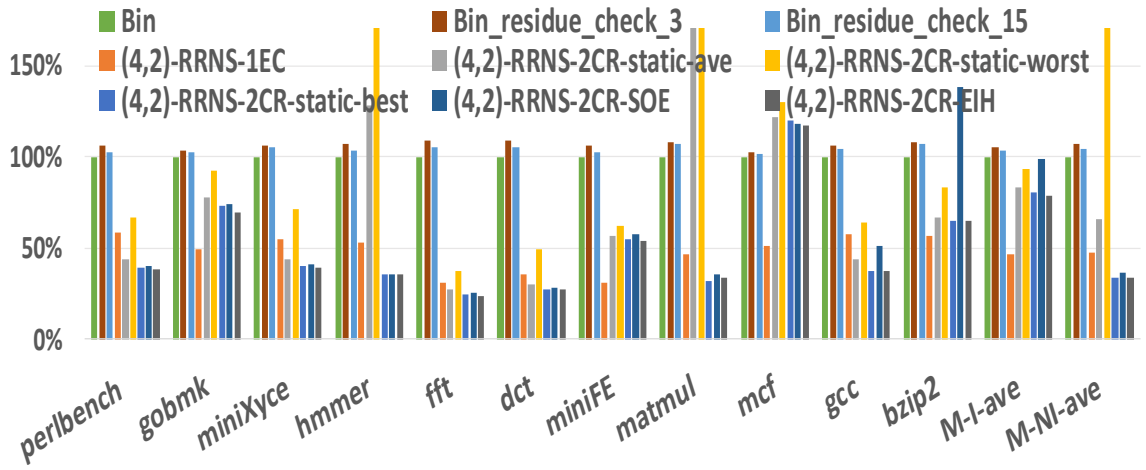


Figure 3.12: Comparison of Checkponting&Restart EDP

cant using techniques introduced in this chapter. In particular, the *EIH* scheme degrades performance by only 11% on average based on non-memory-intensive jobs.

Therefore, the EDP results in Figure 3.12 present a conclusion similar to that of energy. *EIH* is consistently as efficient as or more efficient than *static-best*, and considering that these reductions can be realized without profiling an application to determine its optimal configuration, *EIH* offers significant energy and EDP reductions relative to its integration complexity in the architecture. On average of non-memory-intensive applications, *EIH* achieves an EDP reduction of 67%, 30%, and 1% over Bin, *(4,2)-RRNS-1EC*, and *(4,2)-RRNS-2CR-static-best* respectively.

3.7 Conclusion

A scalable RRNS microarchitecture that simultaneously supports both, error-correction, as well as checkpointing with restart capabilities upon detecting uncorrectable errors is proposed in Chapter 2. At the beginning of this chapter, static RRNS checkpointing&restart mechanisms are designed to verify the potential of this idea. And then, two novel RRNS-based adaptive checkpointing&restart mechanisms that automatically guarantee reliability while minimizing the energy-delay product (EDP). For similar reliability when compared to a conventional binary core without computationally error-tolerant (runs at high V_{dd}), the proposed RRNS scalable microarchitecture reduces EDP by 67% on average for non-memory-intensive workloads. Similarly, *EIH* achieves an EDP reduction of 30% over *(4,2)-RRNS-1EC* on average for these non-memory-intensive jobs.

CHAPTER 4

(N,R)-RRNS CORRECTION-ONLY SYSTEMS FOR MILLIVOLT SWITCHES

From Chapter 3, if a workload is memory-intensive, the system should spend more energy on checkpoint creation, verification, restoration, etc. Thus, correction-only RRNS configurations to avoid the enormous memory footprint tracking are also necessary. The RRNS correction-only methodology contains three consecutive steps: 1) detect possible errors, 2) check error correction Lookup Table (LUT) before proceeding to the next step or complete the correction procedure by directly replacing the wrong residue(s) with the correct value(s) by regenerating them via the Base Extension Algorithm, and 3) add the correction factor(s) from error correction LUT if necessary. An earlier example in Chapter 2 describes this 3-step correction process.

CREEPY[30] is one of the *(4,2)-RRNS* systems supporting only One Error Correction (1EC). However, the *(4,2)-RRNS-1EC* is just a specific configuration of the general *(n,r)-RRNS* correction system. Essentially it explores only a point within the RRNS plane (n and r dimensions) without considering any other configurations. In order to support multiple error correction in RRNS, as discussed in Section 2.3, the number of redundant residues has to be increased. In the correction example of Chapter 2, the `delta_value_pair` is used as the input of error correction LUT, and returns the ID of error residue with a correction factor. Waston [25] concludes the size of error correction LUT grows in an exponential way with the number of correctable errors increasing. In the 1EC mode, accessing the LUT is necessary only when the error is located in one of the non-redundant residues. If the error is found in one of the redundant residues, the correction procedure is to simply replace the incorrect residue with the corresponding Base Extension Algorithm output.

4.1 Size of Error Correction Lookup Table

The size of error correction LUT is a critical consideration of the system's feasibility. Based on the theorems in combinatorics, the amount of possible error cases in the i^{th} non-redundant residue equals to $(m_i - 1)$. m_i is the i^{th} modulus in this RRNS base set and $1 \leq i \leq n$. In the i^{th} residue with modulus m_i, it may change to any of the other $(m_i - 1)$ values and become erroneous. So the total possible error cases that have to access LUT is $\sum_{i=1}^{n}(m_i - 1)$. That is to say, the size of LUT is $K\sum_{i=1}^{n}(m_i - 1)$,where K is the maximum number of entries for each error case. Assuming the size of each entry is 4 bytes with n=4, K=2 and m_i=256, the total size of this single error LUT is roughly equal to 8KB. Similarly, for double error detectable configuration, the size of LUT is $K\sum(m_i - 1)(m_j - 1)$, where $1 \leq i \neq j \leq n$ and $(i, j) \neq (j, i)$. In order to simplify the size estimation, assume m_i = m_j = m = 256, n=4 , K=2 and 4 bytes per entry. So the size of double error correction LUT is $4K\sum(m-1)^2$= 4K $\binom{n}{2}(m-1)^2 \approx$ 3MB. For the tripe error correction, the approximate LUT size is $4K\sum(m-1)^3$= 4K $\binom{n}{3}(m-1)^3 \approx$ 512MB. So the LUT size increases in an exponential manner [25] if raising the number of correctable errors.

4.2 Energy Delay Production (EDP) Comparison

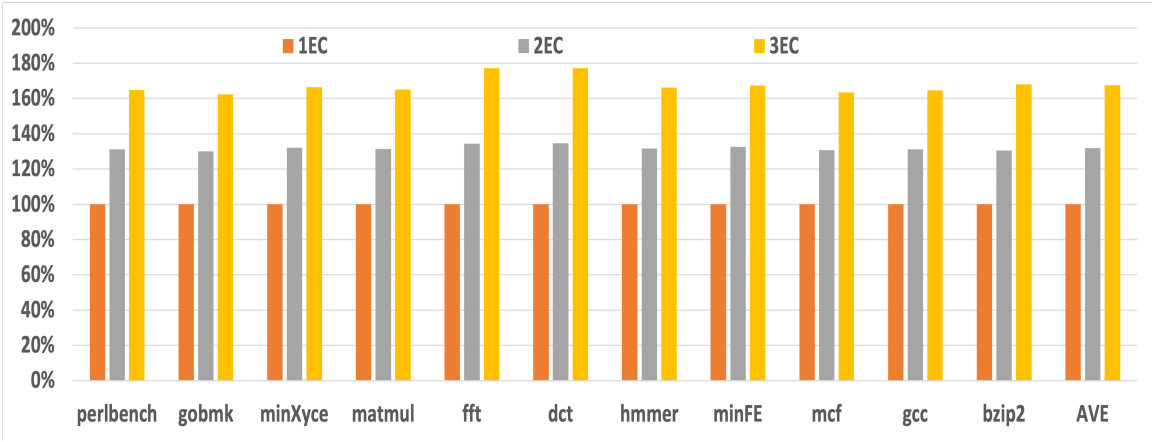


Figure 4.1: Energy Delay Production (EDP) Comparison

A large LUT implies long access latency and a power-hungry design, defeating the purpose of the single error correction design. Figure 4.1 shows the Energy Delay Production (EDP) comparison of 1EC (EC: Error Correction), 2EC and 3EC. The EDP value of 1EC is normalized to 1. It can be seen that as the error correction capability of an RRNS core is increased, the corresponding EDP values also increases significantly. For configurations with stronger correction capabilities, not only are the LUT access delays larger, but also more redundant moduli (subcores in the hardware) are required. Therefore, configurations with better error correction capabilities require more overhead. Compared with 1EC, on average, the EDP values of 2EC and 3EC increase by 31.81% and 67.45% respectively.

RRNS multiple error correction is not a good candidate for lowering EDP, area, and hardware overhead. Therefore, the error correction capabilities of correction-only or hybrid systems should be limited to no more than one error correction.

CHAPTER 5

(N,R)-RRNS HYBRID SYSTEMS FOR MILLIVOLT SWITCHES

5.1 (n,r)-RRNS Hybrid System Design

The scalable RRNS architecture proposed in Section 2.11 simultaneously supports both correction and checkpointing&restart. The first level of resilience is correction while the second lower level is detection with checkpointing&restart. The essential criterion of this hybrid system is if the first level resiliency is insufficient, then the lower level kicks in to offer protection.

On the one hand, this hybrid system may avoid heavyweight recovery and re-execution if the numbers of error residues of all the detected error entries are less than its correction capability. In other words, the probability of complete system recovery and restart may reduce with the help of correction. However, the high overhead correction schemes, such as multiple error correction discussed in Chapter 4, should be avoided. Based on the size consideration of correction LUT, the resilient capability of hybrid systems should be limited to at most one error correctable for a particular RRNS value. On the other hand, the transistor's fault probability is relatively high in the low signal energy environment. Even if only one RRNS value in CC is verified as uncorrectable, the system has to roll back to the last valid machine state. In such a case, all the correction operations of current LI are wasted and further increase total overhead.

5.2 Design Space Exploration of (n,r)-RRNS

This section explores best results for n, r, c and d in an *(n,r)-RRNS-cECdCR* architecture. Section 2.3 discusses the theoretical foundation of *(n,r)-RRNS* design space extension. The scalable error-tolerant RRNS architectures can hence be classified into three categories:

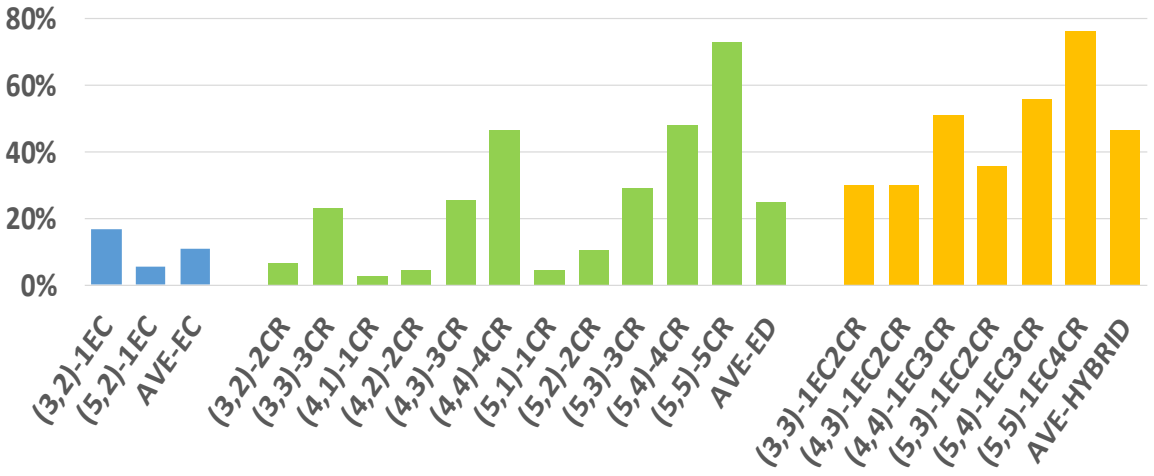


Figure 5.1: Memory-intensive;The energy consumption of RRNS schemes normalized to (4,2)-RRNS-1EC

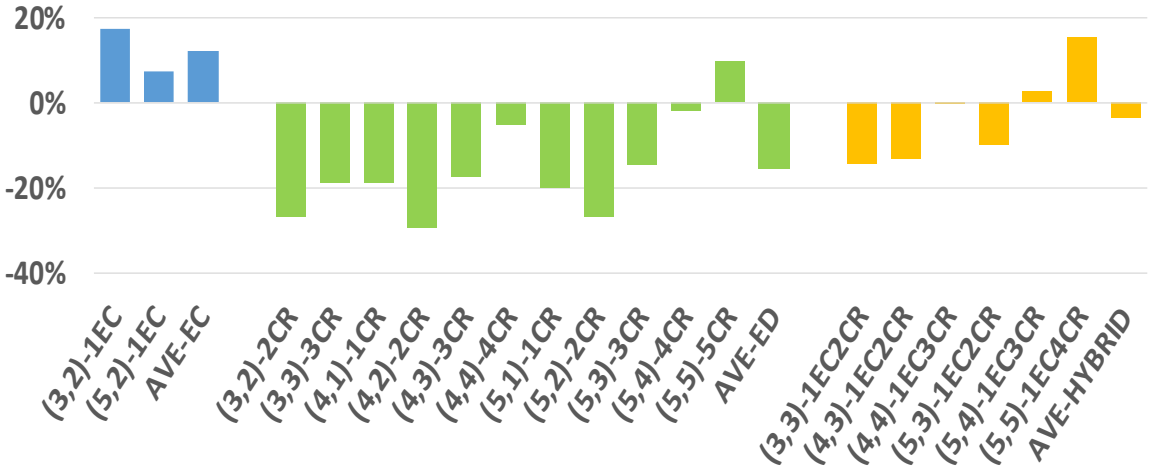
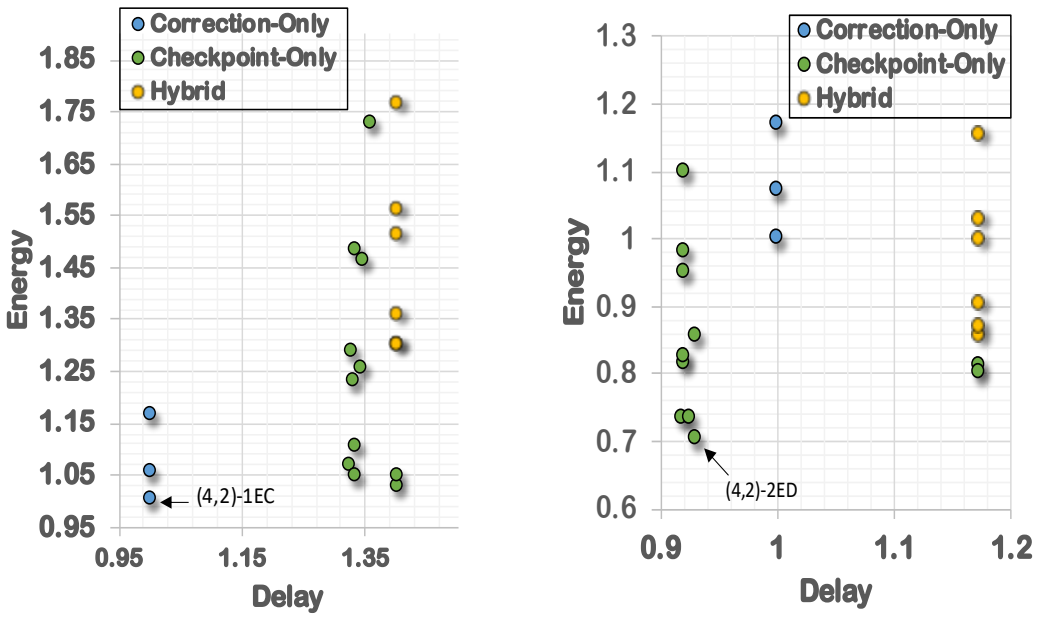


Figure 5.2: Non-memory-intensive;The energy consumption of RRNS schemes normalized to (4,2)-RRNS-1EC



(a) Memory-intensive (b) Non-memory-intensive

Figure 5.3: Energy Delay Pareto normalized to (4,2)-RRNS-1EC

checkpointing-only (Chapter 3), *correction-only* (Chapter 4) and *correction-checkpointing-hybrid* (Chapter 5) systems.

Figure 5.3 plots, normalized to *(4,2)-RRNS-1EC*, energy and delay of various *(n, r)-RRNS-cECdED* schemes. Section 3.6.3 shows that checkpointing/restart is not suitable for workloads that include high-frequency memory operations. Therefore, workloads are divided into memory-intensive and non-memory-intensive for separate discussions. Non-memory-intensive and memory-intensive workloads are represented by perlbench and gobmk respectively in this subsection. Recall that due to the exponential scaling behavior of error correction LUT sizes, the correction based configurations are limited to at most one correctable error. All checkpointing/restart configurations in this subsection use the Error Interval Heuristics (EIH) adaptive method. Similarly, Figure 5.1 and Figure 5.2 show the energy consumption of all the explored configurations.

As shown in Figure 5.1, Figure 5.2 and Figure 5.3, *(4,2)-RRNS-2CR* and *(4,2)-RRNS-1EC* are the configurations which give the best results for non-memory-intensive and memory-

intensive workloads respectively. From Figure 5.3, *(4, 2)-RRNS-1EC* gets the lowest EDP in memory-intensive workload while *(4, 2)-RRNS-2CR* performs best in non-memory-intensive workloads. For the non-memory-intensive jobs, it can be observed that the best *checkpointing-only* scheme with novel mechanisms described in this thesis significantly reduces the total energy (by around 30% when compared with the best *correction-only* configuration), while maintaining similar delay. The hybrid schemes are slightly worse on average because of extra energy overhead while simultaneously supporting both error-tolerant capabilities. Moreover, the error rate of transistors is relatively high in the low signal energy environment. Even if only one RRNS value goes into the undetectable error during the current LI, the system has to roll back to the previous verified machine state. In such a case, the previous corrections of this hybrid system become useless. For memory-intensive jobs, *correction-only* schemes are the best bet. From the perspective of storage overhead, *correction-only* is a lightweight method that avoids the memory footprint tracking.

CHAPTER 6

THREAD-LEVEL FAULT-TOLERANCE FOR EXASCALE COMPUTING

Given all that has been learned about RRNS systems in prior chapters, RRNS can now be applied to challenges of Exascale computing at a thread level.

6.1 Motivation and background

As High-Performance Computing (HPC) complexity grows exponentially and advances towards Exascale, runtime errors become a critical issue for upcoming system designs. Some researchers predict that the Mean Time Between Failures (MTBF) of exascale systems might go down to just several minutes[86, 87]. Moreover, upcoming energy-efficient techniques (i.g, Near Threshold Voltage (NTV) [88] computing) aimed at extending Dennard scaling further increase the system's error rate. The state-of-the-art resilience techniques for HPC have serious shortcomings, including, but not limited to insufficient error detection capabilities, challenging to use, and energy-inefficiency. Instead of adopting a unique fault-tolerance technique, Elliott et al. [89] suggested using a Triple Module Redundancy (TMR) with a checkpointing&restart mechanism for exascale systems. However, this recommendation was made by only considering the tradeoff between reliability and delay.

Orthogonal to reliability, power-efficiency is another urgent challenge of exascale systems[90]. The Green500 List[91], which features power-efficiency, was announced as a supplementary of the TOP500 List. Unlike the Top500 List that highlights TFlops, the Green500 List utilizes GFlops/Watts as the primary evaluation metric [92]. Moreover, the Department of Energy (DOE) Exascale Initiative Roadmap indicates that an exascale system's power constraint should be limited to 20 MW [93, 94, 92], while research reports at around 2010 estimated that such systems needed at least 60 MW even with some idealized assumptions [90, 95]. Frontier[96], an exascale computer (theoretical peak

performance>1.5 exaflops) jointly developed by Cray, AMD, and Oak Ridge National Laboratory, is expected to be delivered in 2021. The power consumption of Frontier has been estimated as 29 MW[97]. Despite ten years of technical upgrades and improvements, there is still a significant gap between the DOE threshold and this state-of-the-art estimated value. Therefore, systematic explorations for power optimization of exascale systems still needs to continue.

This chapter presents the microarchitecture of an in-order pipelined core for exascale systems relying on the Redundant Residue Number System (RRNS) to support thread-level fault-tolerance and significantly reduce power consumption. By analyzing the working principles of computational error-tolerant techniques, it could reach the conclusion that RRNS has more advantages than traditional Triple Modular Redundancy (TMR) [98] in terms of power consumption and computational logic area. Since power consumption and reliability are now critical for exascale system design, RRNS concepts could be integrated into exascale system designs to limit their power consumption [95]. The design and optimizations presented in this chapter can achieve sufficient system reliability with lower power consumption to bring exascale systems closer to the efficiency threshold proposed by DOE.

CREEPY and related improvements [29, 7, 30, 31, 26] is an instruction-level RRNS core design, which implies that all the subcores in a CREEPY core execute the same instruction at a specific timestamp. CREEPY's instruction behavior is similar to Single Instruction Multiple Data (SIMD) from Flynn's taxonomy [99], and the synchronization interval among residues/subcores is only a single instruction. In this chapter, the constraint of executing an identical instruction among all subcores is removed and establishes an asynchronous thread-level mode similar to Multiple Instruction Multiple Data (MIMD). Thus, allowing a reduction in system performance degradation introduced due to RRNS integration. Moreover, this thread-level RRNS resiliency model can be easily transplanted to existing exascale programming models.

This chapter presents a thread-level Redundant Residue Number System (RRNS) scheme and discusses the corresponding microarchitecture design by following the unique execution mode of thread-level RRNS. This allows RRNS to be efficiently applied to exascale systems improving their fault-tolerance and energy-efficiency. It also shows the RRNS API compatible with the current Habanero C/C++ library (HClib)[27], which demonstrates the feasibility of thread-level RRNS in the current Asynchronous Many-Task (AMT)[28] programming model widely used for programming exascale systems. Through further optimizations of thread-level RRNS microarchitecture, this method shows 62.25% and 58.67% reduction respectively in energy and Energy Delay Product (EDP), compared with the state-of-the-art Asynchronous Many-Task (AMT) black-box resiliency method.

6.2 Asynchronous Many-Task (AMT) Programming model

Resilience is projected to be one of the urgent challenges in achieving exascale computing and beyond, mainly due to the higher fault rates in such systems compared to current designs. These faults could be classified into two main categories [100]: fail-stop and fail-continue. Fail-stop faults directly cause the compute nodes to stop or crash, which results in the loss of all computational states and data. Typically, such failures are detected by the operating system or middleware used in HPC systems. On the other hand, fail-continue faults often caused due to soft/transient errors can cause the process to fail, but still continue to execute. There are some cases where these soft/transient errors go undetected by the conventional hardware, operating system, and other middleware components. Such errors are called Silent Data Corruptions (SDC) [101]. Even when such errors are detected, the process might not be able to correct them and even continue execution (Detected Uncorrectable Errors - DUE) [101]. Faults due to soft/transient errors are the more critical issue when compared to fail-stop faults in exascale systems [100].

The Asynchronous Many-Task (AMT) is a category of programming and execution models proposed as an alternative to the dominant bulk synchronous programming mod-

els. In the AMT programming model, an application program is decomposed into small, transferable units of work called tasks with associated data inputs rather than directly decomposing at the process level such as MPI [102] ranks. AMT foundations of transferable task units have shown promising potential [103, 104, 105, 106, 107] in mitigating the effects of fail-continue faults. To enable resilience, it is necessary to identify the program location and data to perform error checking and recovery. In AMT programming models, the task boundary provides an ideal program location around which can directly implement resilience. The data that is passed across the task boundary, i.e., the task outputs, gives us the targeted data that needs to be checked to ensure correctness.

Before discussing the fault-tolerance of AMT models, the types of resilient techniques should be defined and classified. Similar to the testing classification, according to the user's understanding of the application/algorithm, the resilient techniques can be divided into black-box resiliency and white-box resiliency. Black-box resiliency means that the users do not need to understand the target application/algorithm deeply, and they just use one or more simple APIs to denote which program segments need the resilient technique's support. In contrast, white-box resiliency requires users to have an in-depth understanding of the target application/algorithm and provide one or more user-defined error detection/correction methods to achieve fault-tolerance.

Paul *et al.* [106] demonstrated a comprehensive approach to achieve resilience in AMT programming models with a focus on remedying Silent Data Corruption (SDC) on shared memory systems that often go undetected by the conventional hardware, operating system, and middleware. Different resilience techniques, including task replay which includes replaying a task in case of a failure, task replication which includes creating multiple copies of a task to compare their results, algorithm-based fault tolerance (ABFT) which includes application-specific error correction provided by the user, and checkpointing which allows restoring data from a previous checkpoint in case of failure, are demonstrated in this work. They use Habanero C/C++ library (also known as HClib) as an exemplar AMT runtime

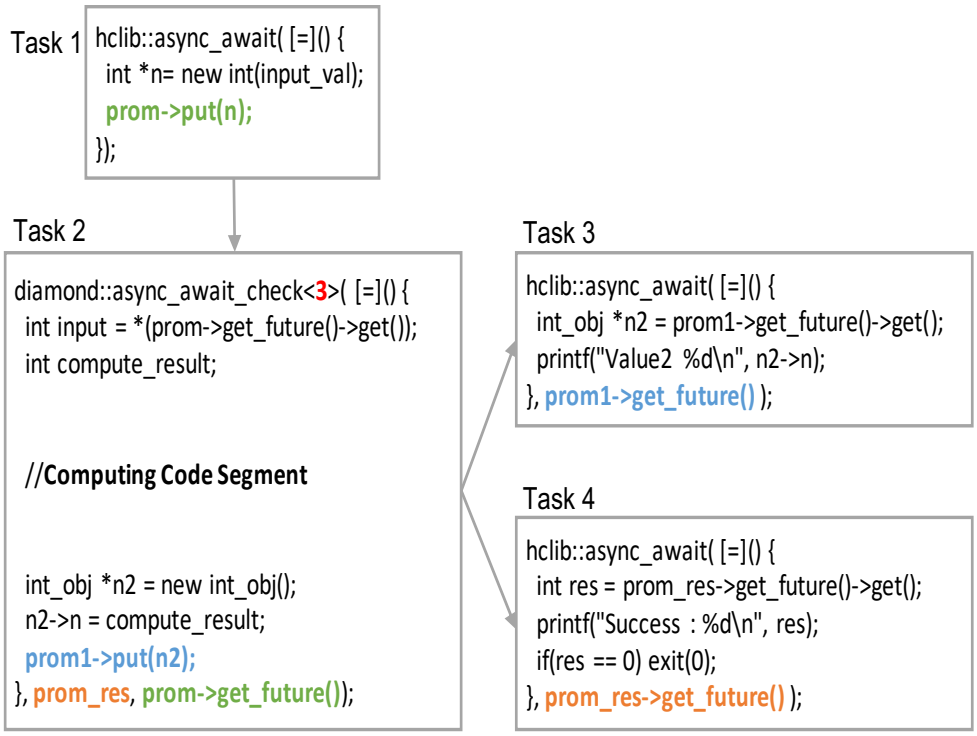


Figure 6.1: TMR-AMT Example

for this purpose. Paul *et al.* [107] further extended the resilient HClib runtime from [106] by adding support for communication across nodes using the MPI communication library. This work [107] demonstrates how to tolerate fail-stop faults along with silent data corruption by adding support for transparent recovery from a node crash without any user intervention.

Now let us analyze the various resilience techniques described above in terms of energy efficiency and ease of use. The task replication resilient technique is similar to Dual/Triple Module Redundancy(D/TMR), which has been demonstrated as energy-inefficient in Section 6.4. The remaining first-level resilient methods (task replay and ABFT), require the user to understand the algorithm/application and implement application-specific resiliency case by case. That is to say, extra algorithmic research works are needed to get reliability. The state-of-the-art resilient techniques in software AMT models are either energy-inefficient (task replication, which offers black-box resiliency) or hard to use (task replay and ABFT, which provide white box resiliency).

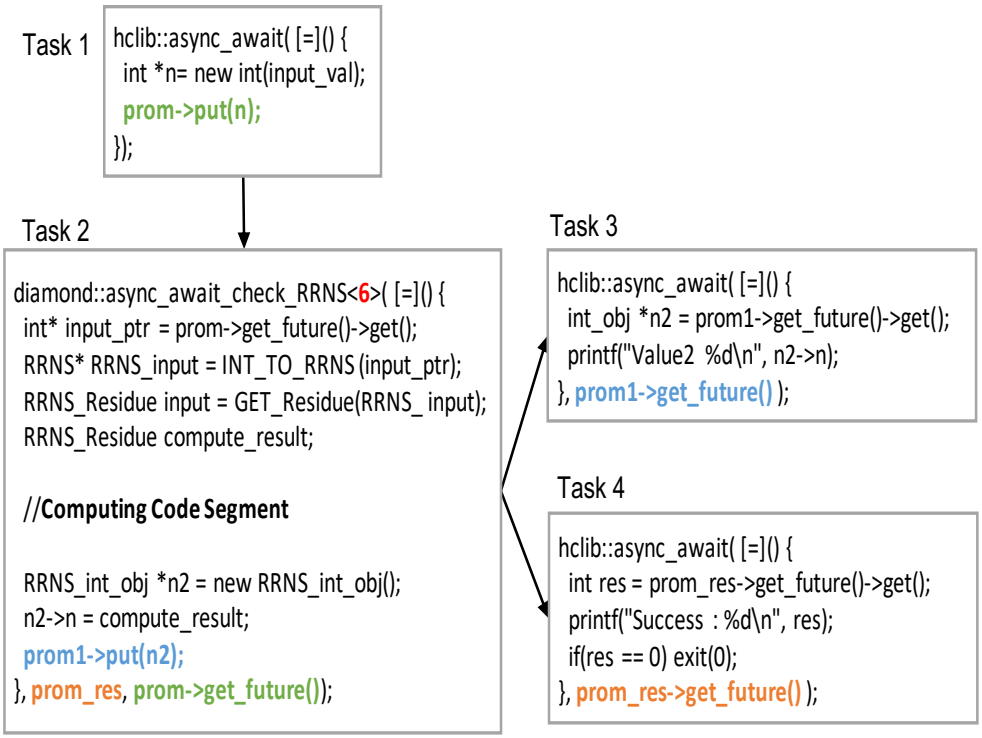


Figure 6.2: RRNS-AMT Example

Figure 6.1 shows an example of implementing task replication through HClib APIs [27]. ***async_await*** API calls are used to create asynchronous tasks. The executing body of each task is represented by a user-defined C++ lambda expression. The dependency between asynchronous tasks is defined by the Promise & Future pair [108] which was introduced in C++11, such as the ***prom→put(n)*** and ***prom→get_future()*** marked in green. Figure 6.1 shows an example of the Triple Modular Redundancy (TMR) resilient technology, which is implemented through the resilient task creation API ***async_await_check*** in Task 2. Task-level TMR uses two extra tasks internally to run the same computation. When these three subtasks are completed, ***async_await_check*** performs equality checking on the three internal subtask's results that were added to ***prom1*** and satisfies this promise on success, subsequently scheduling task 3.

The equality checking can also fail, in which case, that information needs to be communicated with the user. This information of success(1) or failure(0) is reflected in the ***prom_res*** parameter which activates Task 4 once the equality checking finishes. More de-

tails about HClib and AMT replication are available in [106].

A suitable RRNS setup is theoretically more energy/area efficient than TMR for computational error detection and correction. This could be achieved via augmenting the existing HClib framework with RRNS through a similar TMR-AMT method and modify the corresponding APIs, as shown in Figure 6.2. The library needs to create a new ***async_await_check_RRNS*** API, which can adjust the number of lightweight threads according to the RRNS configuration. For example, the configuration is (4,2)-RRNS, which can support six lightweight subtasks, and each task processes the corresponding residue. In other words, the bit-width of a lightweight task is the same as its corresponding residue, such as 8/9 bits. The TMR method's task is a complete copy of the original computation (such as 32 bit), not a lightweight task unlike RRNS. When ***async_await_check_RRNS*** is completed, the AMT runtime will collect all the six residue results added to the promise ***prom1*** and performs RRNS error checking on them. Once it succeeds, the runtime will use a user-provided function to assemble the residues and convert them to ***int_obj*** and use it to satisfy the promise, which schedules Task 3 for execution. Thus, this should be able to transplant thread-level RRNS to the existing AMT programming model with minimal changes to the existing APIs through this proposed method.

6.3 Thread-level RRNS Resiliency

6.3.1 RRNS-AMT Overview

For multithreading programming models, such as AMT, the input data is converted from conventional binary format to RRNS format and maps the corresponding residues to inputs of a group subtasks (low-bitwidth threads) to achieve reliability.

Similar to the example in Table 1.1, an execution overview example shown in Figure 6.3 also adopts a $(4, 2)$-RRNS configuration. The first four threads are logically grouped together as non-redundant threads, while the remaining two are called redundant threads. However, all six threads are low-bandwidth threads. That is to say, the bit width of these

threads is typically 8/9-bits, equal to the bit-widths of practical RRNS moduli, instead of the usual 32/64-bit threads found in Dual/Triple Modular Redundancy(D/TMR). Each of these lightweight threads asynchronously executes the same set of micro-instructions, i.e., the same opcodes issued in sequential order with each thread handling a different residue of the RRNS data. For example, the 2^{nd} thread of the group only uses the 2^{nd} residue of a particular RRNS data as input, and its output only contributes to the 2^{nd} residue of the RRNS output. When an RRNS unfriendly instruction, such as a comparison operation, is encountered, the threads are synchronized similar to a barrier operation, since the RRNS unfriendly instruction needs all its corresponding residues to be available before it can be executed.

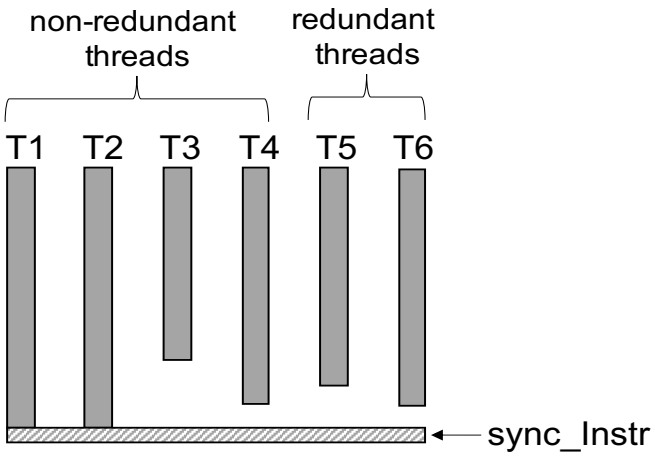


Figure 6.3: (4,2)-RRNS-AMT Overview

When the system or the user needs to perform error detection, each thread sends its residue of the corresponding RRNS value to the Residue Interaction Unit (RIU). The RIU is a centralized hardware unit that handles RRNS synchronized operations, including RRNS to binary conversion, fractional multiplication, and error detection/correction logics. Synchronizing threads for error detection is unnecessary because RRNS error detection occurs off of the critical path of execution. A simple auxiliary register waits for the remaining residues of an RRNS instruction that have not arrived. The RIU cannot proceed until all six residues of a corresponding instruction are ready. If an error is detected, the correction

or recovery&re-execution procedure is called according to the system configuration. If no error is detected, the execution of the critical path proceeds uninterrupted.

6.3.2 Thread-level RRNS Limitations and Solutions

The RRNS-AMT mode relaxes unified instruction constraints by allowing all RRNS sub-cores to operate independently on multiple instructions. However, an RRNS unfriendly instruction such as division, bit manipulation, or comparison needs a barrier like synchronization for further processing via the Residue Interaction Unit. From experience with parallel programming, it is known that frequent insertion of barriers for thread synchronization improves significant overhead and might seriously degrade the performance of exascale systems. Therefore an effective method for minimizing the number of barriers generated by RRNS unfriendly instructions needs to be proposed.

Table 6.1: Percentages of RRNS unfriendly instructions

	Cmp	**Bit-Op**	**Div**	**Sqrt**
perlbench	11.46%	2.52%	<0.01%	0.00%
gobmk	18.72%	0.17%	0.00%	0.00%
miniXyce	9.33%	1.95%	0.06%	0.01%
hmmer	12.38%	1.27%	0.15%	0.00%
fft	2.72%	<0.01%	<0.01%	<0.01%
dct	2.44%	<0.01%	<0.01%	0.00%
matmul	13.58%	0.12%	<0.01%	0.00%
miniFE	11.23%	0.17%	0.03%	<0.01%
mcf	15.98%	0.49%	0.00%	0.00%
gcc	12.03%	3.05%	<0.01%	0.00%
bzip2	12.87%	4.05%	0.00%	0.00%
average	11.16%	1.25%	0.02%	<0.01%

Table 6.1 summarizes the proportions of RRNS unfriendly instructions in each of the workloads (from the SPEC, Mantevo Miniapp Suite, and user-defined benchmarks) evaluated in this chapter. The most straightforward conclusion from this instruction breakdown is that *comparison (cmp)* instructions are the most significant contributor to RRNS barrier synchronizations. On average, *cmp* instructions account for 11.16% of all instructions,

while the sum of the remaining three categories is less than 1.28%. Therefore, reducing the frequency of *cmp* induced synchronizations in the critical path is the primary prerequisite for optimizing thread-level RRNS performance.

Systematic profiling of the ARM assembly traced used for evaluations shows that on average, 68% of *cmp* instructions are closely related to branches. More specifically, this category of *cmp* instructions is directly followed by a related branch which uses the result from the *cmp* instruction to determine its jump direction. The *cmp* instruction sets the Current Program Status Register (CPSR), which the branch uses to decide if it should be taken or not. By following this observation, an efficient methodology is proposed to reduce the synchronization frequency of RRNS barriers by simple modifications to the branch predictor's input. Typically, when a `cmp` instruction is detected in the decode stage, the pipeline is immediately stalled, and the `cmp`-related residues are forwarded to the RIU. However, this is equivalent to adding a long synchronization stall on the critical path of execution, which has the potential for adverse effects on exascale systems' performance. On the other hand, this optimization checks if the instruction in the pipeline's fetch stage is a corresponding branch. If the following instruction is a branch, it can safely remove the *cmp* in the decode stage from the critical execution path because the purpose of this *cmp* is to set the value of status registers for the upcoming branch. The branch predictor's result is used to determine the branch's behavior allowing execution to proceed without stalling, providing the opportunity to execute the `cmp` instruction outside the critical path. When the result of the `cmp` is available from the RIU, the branch predictor's correctness is verified. If the branch is mispredicted, the branch misprediction process is followed: the pipeline is flushed, status recovered and the program is re-executed right after the mispredicted branch. This optimization is called the *Branch Predictor Combination*.

The Program Counter (PC) adder in the fetch stage is an orthogonal constraint that may hurt the system's energy efficiency. The intuitive design of a thread-level RRNS core is to include the pipeline's Instruction Fetch stage in a closed subcore according to the cor-

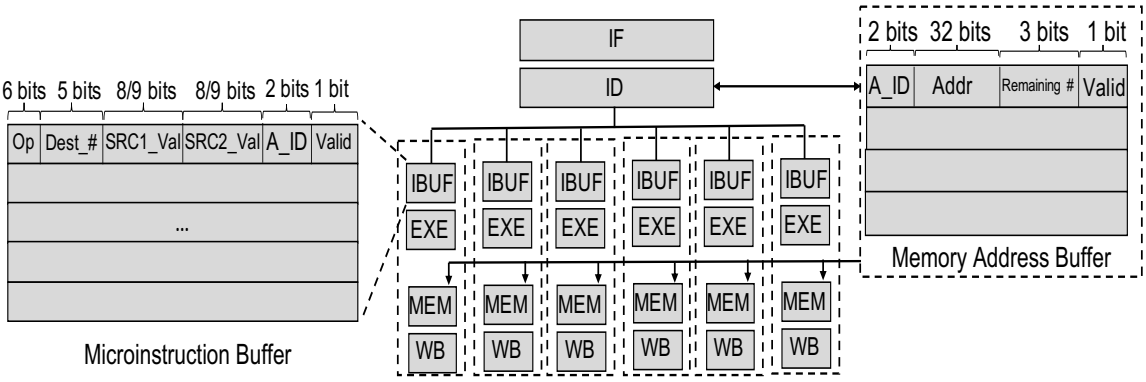


Figure 6.4: The Resilient Microarchitecture Overview of Thread-level Redundant Residue Number System

responding residue's bit-width. From this intuitive design, the fetch stage of each subcore must contain an independent PC adder. For example, if the core uses a 6-residue configuration, five extra PC adders are required to support thread-level RRNS. This undoubtedly goes against the goal of lowering power consumption. Moreover, these additional adders also increase the core area, making it paramount to remove the requirement of additional adders in RRNS subcores. To accommodate this constraint, a shared fetch stage is designed and optimized such that only one PC adder is required. A simple micro-instruction buffer is added per subcore and the pipeline is augmented with an IBUF stage between the Instruction Decode and Execute stages. More details of this augment and how these are leveraged to achieve efficient thread-level RRNS are detailed in Section 6.3.3.

6.3.3 RRNS-AMT Microarchitecture

This section discusses the CPU microarchitecture details of the thread-level RRNS core that supports the RRNS-AMT. Figure 6.4 illustrates the overview of the microarchitecture of a pipelined processor featuring six RRNS subcores where each subcore corresponds to one residue and four cores are non-redundant subcores, and the remaining two are redundant subcores used to detect potential errors. In the orthogonal direction, this microarchitecture has a six-stage in-order pipeline. Five stages of this six-stage design correspond to stages in the classic five-stage MIPS pipeline, while a new IBUF stage is added between the ID

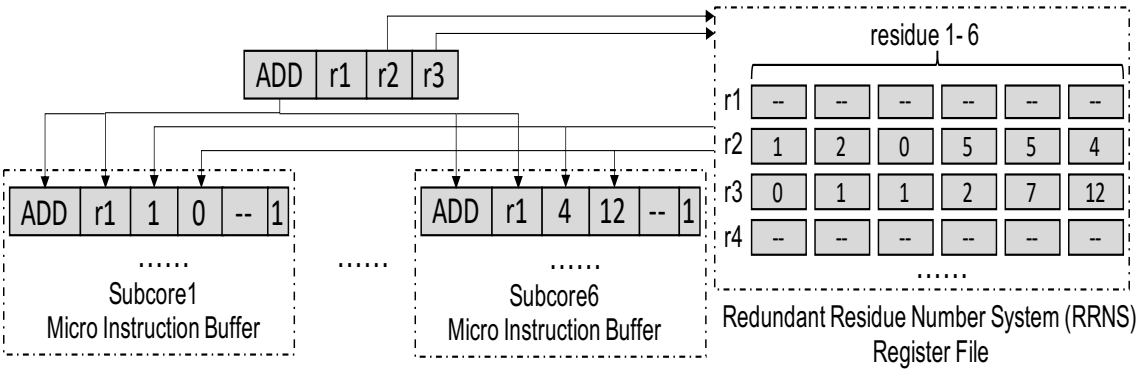


Figure 6.5: An Example of Converting An ADD Instruction Into Multiple ADD Micro-instructions In The Decode Stage

and EXE stages.

The IBUF stage contains a Micro-Instruction Buffer (MIB) per subcore, and the size of which is between 5-10 instructions per the system design requirements. If a valid un-executed micro-instruction is available in this buffer, the corresponding subcore fires it according to resource availability. The MIB of each subcore is completely isolated in the horizontal direction, i.e., it does not interfere with MIBs of other subcores.

The translation from the original instructions to the micro-instructions are processed and assembled in the Instruction Decode (ID) stage, and then forwarded to the MIBs of the relevant subcores based on their residue order. The instructions in the MIB are tracked via two pointers, one each at the beginning and end of valid instructions (`beg_ptr` and `end_ptr`), to form a circular buffer, rendering it unnecessary to move instructions around when inserting or deleting micro-instructions from the MIB. On insertion, the location following the current end pointer (`end_ptr+1`) of the MIB is checked, and if the valid bit of that location is set to 0, the micro-instruction is inserted and the end pointer incremented. If the valid bit of location `end_ptr+1` is set to 1, it indicates that this subcore has no free slots because the micro-instruction presently in that slot is still waiting to be issued. When this happens, the pipeline's fetch and decode stage are immediately stalled. The two stages cannot resume execution until each the subcore finds an unoccupied location in its MIB.

In this processor, both the Register File (RF) and the cache hierarchy store data in

the RRNS format. When an instruction reaches the decode stage, the system reads the necessary source operands from the RF, and the corresponding slice-values (residues) are forwarded to the relevant fields of subcores' micro-instruction buffers. Figure 6.5 is an example of an ADD (`ADD R1, R2, R3`) instruction being converted into multiple ADD micro-instructions in the decode stage of the pipeline. Similar to the decode mechanism in a traditional pipeline, the first source register (`R2`)'s contents are read, but the residue values from RF are distributed to the subcores, the first residue value to *subcore1* and so on and so forth. The second source register's (`R3`) contents are also read and distributed in a similar manner at the same time. When all the necessary fields in the MIB entry are set (*Opcode, Dest_#, SRC1_Val, and SRC2_Val* in the case of `ADD`), the status bit is set to 1 (Valid) and the micro-instruction is ready to be fired.

Load and Store instructions need to access the cache hierarchy in the MEM stage of the pipeline. However, in thread-level RRNS, this process is not straightforward since each subcore contains a fraction of the operation's memory address and it is impossible for a single thread to figure out the corresponding complete memory address by relying solely on its own fraction. Assembling complete memory addresses requires synchronizing threads at every memory operation by adding a barrier overhead like other RRNS unfriendly instructions, which is really prohibitive to performance. Another intuitive approach is to allocate a complete memory address field in each micro-instruction (i.e., per core). However, this would significantly increase the MIB's size reducing energy gains due to RRNS, and therefore is not a good solution.

To effectively conquer the memory operation address problem, a lightweight Memory Address Buffer (MAB) is added for maintaining active memory addresses, as shown in Figure 6.4. This solution has a smaller overhead than storing the entire memory address in the micro-instructions. A new entry is allocated in the MAB whenever a memory instruction is encountered in the Instruction Decode stage and there is available space. Each MAB entry includes four fields: a memory address ID, (`A_ID`) , the complete memory address (`Addr`),

a counter to track the number of threads that still require the address (`Remaining #`), and a valid bit for the entry. The address ID is forwarded to each subcore and added to the MIB entry of the memory instruction as part of the micro-instruction of that core. The usage counter is initially set to the number of threads in the RRNS core to represent the number of times the entry will be used before it becomes invalid. For example, the usage counter is set to 6 (`Remaining #=6`) for the RRNS architecture in Figure 6.4. Whenever a subcore reads an entry from the MAB, the (`Remaining #`) counter is reduced by 1. When an entry in the MAB's (`Remaining #`) reaches 0, it can be reclaimed for reuse and its valid bit is set to 0. As you will notice, this scheme is conservative in nature and allocates a new entry in the MAB even if two memory instructions map to the same complete address. This scheme's resource allocation relies on the instruction stream rather than on unique memory addresses, and from experience, a MAB between 5-10 entries depending on the workload's nature was found to be adequate. Please note that when the MAB has no available slots, the pipeline is stalled until any presently executing memory instructions finish and an entry is freed.

To summarize, this section presents the detailed core microarchitecture of a processor that supports thread-level resiliency via the use of RRNS. However, this does not imply that the entire exascale system needs to use this core in order to attain reliability. This type of core is intended as a small subset distributed in heterogeneous exascale systems where reliability and energy expenditure are top-level constraints along with good performance characteristics.

6.4 Evaluation Methodology And Experimental Results

This chapter aims to provide an easy-to-use black-box fault-tolerant methodology that meets low energy consumption requirements to solve exascale computing's current and future critical challenges of reliability. While white-box resiliency methods such as replay or Algorithm-Based Fault Tolerant (ABFT) may offer alternate solutions to resiliency,

they require user-defined error detection algorithms and a deeper understanding of the underlying applications. This makes white-box techniques less versatile and harder to use. Moreover, the complexity and overheads of error detection vary from algorithm to algorithm and depend on their specific implementations. Therefore, this chapter's proposed methodology compares to other black-box resiliency techniques that meet the versatility and ease-of-use criterion. The section primarily discusses and analyzes three evaluation aspects: performance delay, energy consumption, and Energy-Delay-Product (EDP) of each methodology.

As discussed in Section 6.3.2, RRNS unfriendly instructions require mandatory synchronization of all residue threads, thus reducing system performance and energy efficiency. Although the Branch Predictor Combination (comparison+branch) reduces the total number of synchronizations to a certain extent, these RRNS-related synchronizations cannot be completely eliminated. Due to the need for mandatory synchronization of such unfriendly RRNS instructions, simulations are unable to be performed directly on the HClib[27] platform. Because the use of HClib API is directly oriented to programmers, the minimum schedulable granularity is a single C/C++ instruction, not a single assembly instruction required by the thread-level RRNS. Moreover, the HClib platform does not support the corresponding energy evaluation model. Based on the above requirements, a cycle-accurate trace-driven simulator is designed to model the thread-level RRNS schemes. Gem5 [109] is used as the simulation front end, generates ARM assembly traces through its trace-based debugging method, and then using these traces as the simulator's inputs. These simulation traces are collected from Mantevo Miniapp Suite, SPEC2006 Suite, and user-defined workloads. Energy evaluation results are obtained by integrating each module's signal energy, estimated transistor counts, error detection frequency, and other corresponding factors.

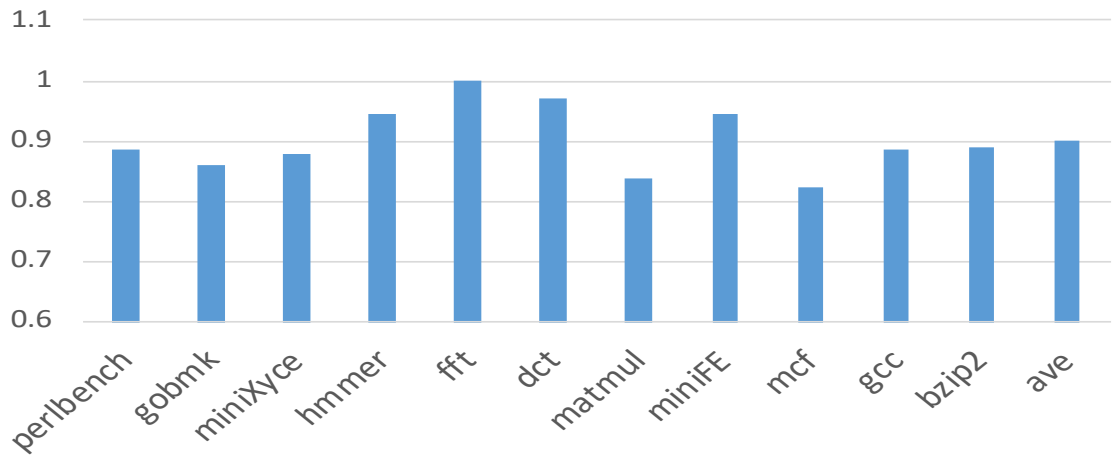


Figure 6.6: Performance Improvement (Delay Reduction) of Branch Predictor Combination

6.4.1 Benefits of Branch Predictor Combination

In Section 6.3.2, it can be observed that comparison instructions take up the vast majority of RRNS unfriendly instructions in the evaluated workloads. Based on this interesting observation, a modification to the CPU microarchitecture is needed where a comparison instruction and its related (following) branch are combined as an input to the branch predictor, thereby reducing the frequency of RRNS comparison synchronization in the critical path of program execution. This section compares the Branch Predictor Combination method with the intuitive design to see how much system performance improvement can be achieved via adopting this optimization.

Figure 6.6 shows the delay comparison between the proposed Branch Predictor Combination (subsection 6.4.1) optimized and the intuitive implementation which adds barrier like synchronization for every comparison instruction. The delay on the x-axis has been normalized to the conventional implementation. The optimization is able to reduce the delay in the best case scenario (`mcf`) by 17.69% while in the worst-case scenario (`fft`) by 0.02%. `fft` does not get a significant benefit from the Branch Predictor Combination since its fraction of comparison instructions is quite low (2.72%) and out that only 0.69% are followed by a related branch. On average, the Branch Predictor Combination Optimization is

able to reduce the delay by 9.82%, improving the usability of thread-level RRNS.

6.4.2 Error-free Scenarios

This section presents a comparison between two black-box techniques for resiliency, the RRNS-AMT approach and TMR-AMT, for the error-free scenario along with the delay, energy and energy-delay product metrics. The error-free scenario implies that no errors are generated nor injected during execution. However, the system's cost of error detection is still must be paid. The orthogonal error-prone scenario is discussed later in section 6.4.3. The error detection frequencies in the evaluation range from every hundred instructions to every million instructions. Figures 6.7, 6.8, and 6.9 summarize the performance of RRNS-AMT normalized to TMR-AMT for delay, energy and EDP metrics. Because TMR-AMT only has the ability to perform Single Error Correction (SEC), the RRNS-AMT core is limited to a $(4, 2)$-RRNS configuration, which according to [110] only has the one error correction ability, for a fair comparison. Following on, unless otherwise stated, all RRNS-AMT results are presented using the $(4, 2)$-RRNS configuration.

Delay: Figure 6.7, the delay results, shows that TMR performs better than RRNS across the benchmark applications. This is attributed to the prevalence of RRNS unfriendly instructions. For example, every bit-operation, such as bitwise AND, requires the system to first convert the RRNS operands to binary, perform the computation, and then convert the result back to RRNS format. Further, RRNS error detection requires extra processing compared with TMR, since RRNS sends residues to the RIU for consistency checks, while TMR only requires a simple majority vote on the output. However, the performance degradation of RRNS is quite limited, between 20.73% (gobmk) in the worst case and 0.55% (dct) in the best case scenario. On average, RRNS suffers from a 10.24% increase in delay compared to TMR when checks are made every million instructions for both.

Energy: Figure 6.8 compares the energy consumption between RRNS and TMR AMT. $(4, 2)$-RRNS has significantly lower energy consumption when compared with TMR across

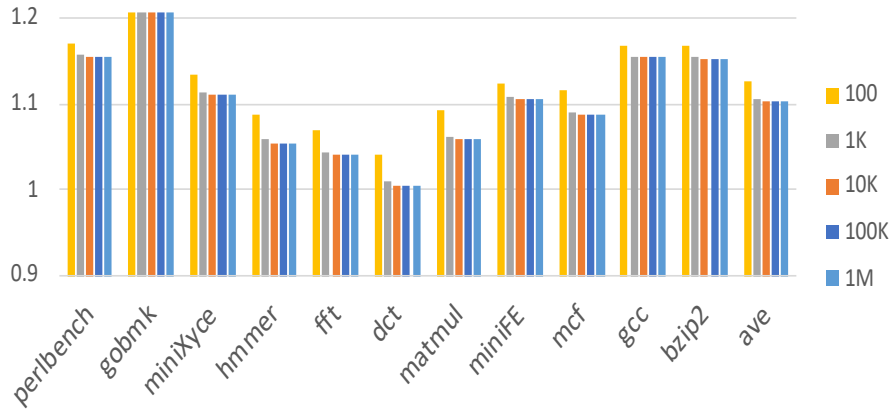


Figure 6.7: Delay of Thread-level Redundant Residue Number System (RRNS); The values in this figure are normalized to Triple Modular Redundancy (TMR)

all configurations. The primary reason for this energy reduction is RRNS remarkable advantage in total computational bit-width and subsequently area compared with the redundancy in TMR. Additionally, RRNS energy consumption can be further reduced for multiplication-intensive workloads by optimizing RRNS multipliers using the index-sum multiplication techniques. `fft` sees the best energy reduction of 79.44% while the worst, `mcf` still has an energy reduction of 49.80% when compared with TMR. On average, $(4, 2)$-RRNS with an error detection frequency of one million instructions lowers energy consumption by 63.54%.

EDP: The energy delay product (EDP) further reveals the tradeoff between energy and delay for each scheme. Figure 6.9 shows that on average, the EDP of $(4, 2)$-RRNS with an error detection frequency of one million instructions is 59.80% compared to the Triple Modulo Redundancy AMT scheme. That is to say, RRNS sacrifices 10.24% delay in exchange for 63.54% energy reduction. Considering the critical need for energy reduction in exascale systems, RRNS provides a compelling case for its use.

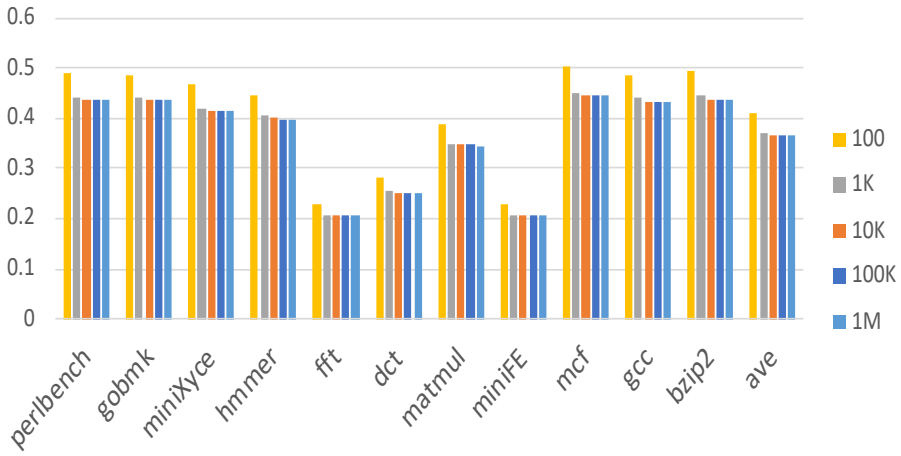


Figure 6.8: Energy of Thread-level Redundant Residue Number System (RRNS); The values in this figure are normalized to Triple Modular Redundancy (TMR)

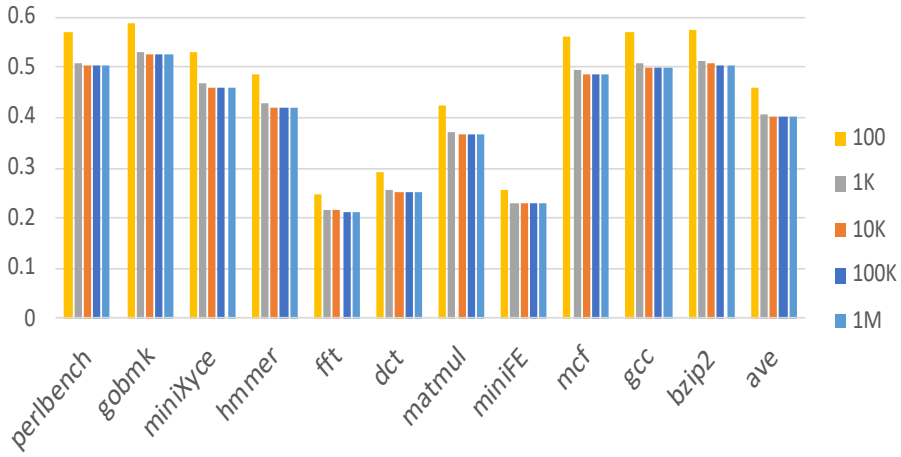
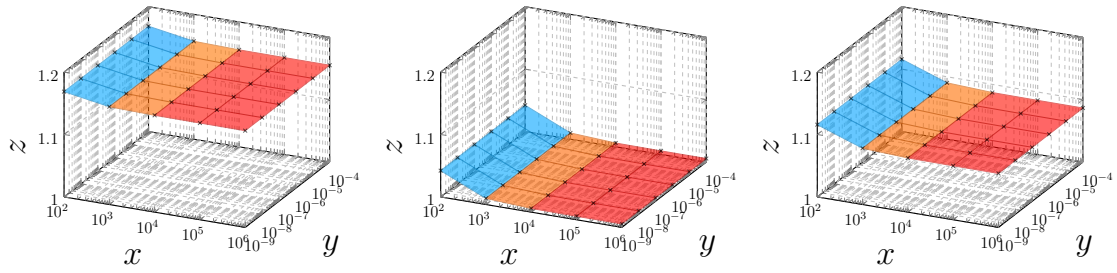
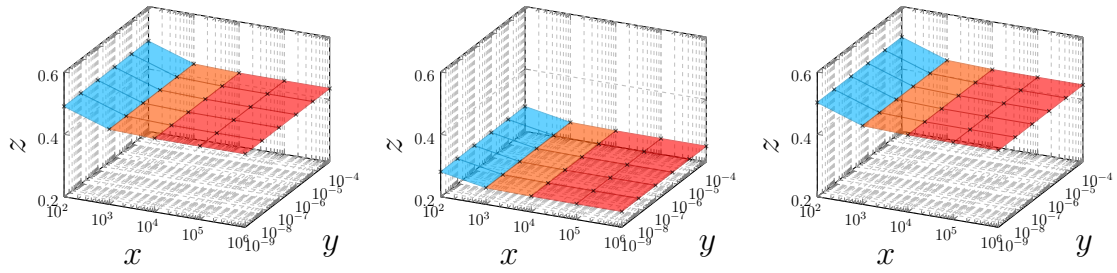


Figure 6.9: Energy Delay Product (EDP) of Thread-level Redundant Residue Number System (RRNS); The values in this figure are normalized to Triple Modular Redundancy (TMR)



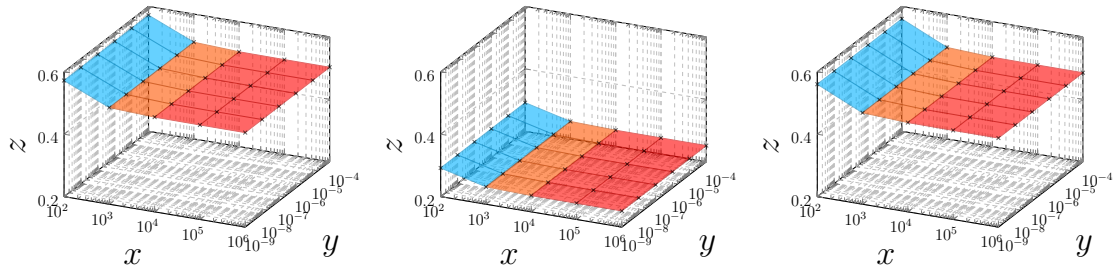
(a) perlbench (b) dct (c) mcf

Figure 6.10: Normalized delay results for different error detection frequencies and error rates; X-Axis: error detection frequencies range from 10^2 to 10^6; Y-Axis: instruction error rates range from 10^{-4} to 10^{-9}; Z-Axis: Normalized thread-level RRNS delay (TMR results are normalized to 1).



(a) perlbench (b) dct (c) mcf

Figure 6.11: Normalized energy results for different error detection frequencies and error rates; X-Axis: error detection frequencies range from 10^2 to 10^6; Y-Axis: instruction error rates range from 10^{-4} to 10^{-9}; Z-Axis: Normalized thread-level RRNS energy (TMR results are normalized to 1).



(a) perlbench (b) dct (c) mcf

Figure 6.12: Normalized EDP results for different error detection frequencies and error rates; X-Axis: error detection frequencies range from 10^2 to 10^6; Y-Axis: instruction error rates range from 10^{-4} to 10^{-9}; Z-Axis: Normalized thread-level RRNS EDP (TMR results are normalized to 1).

6.4.3 Error-prone Scenarios

In this section, RRNS-AMT and TMR-AMT are compared under the error-prone scenario. In order to present a comprehensive evaluation and take into account the possible deviation in predicted Mean Time Between Failures (MTBF), the instruction error rates are conservatively set between 10^{-6} and 10^{-9}. The error checking/detection frequency is varied between 100 instructions and one million instructions similar to section 6.4.2. If the reciprocal of total instructions in a benchmark is greater than the corresponding error-rate for a specification, then the execution is deemed error-free. Conversely, assuming that the errors are distributed to different data entries if more than one error is to be injected in a given interval. Within this error model, both $(4, 2)$-RRNS and TMR can directly correct each data entry, one after another. Otherwise, it is necessary to use a more complex mechanism such as checkpoint plus restart as the second level of resiliency to ensure the system's correctness. A comprehensive resiliency scheme using checkpointing is not discussed in this chapter.

Figures 6.10, 6.11, and 6.12 compare the delay, energy and EDP of $(4, 2)$-RRNS and TMR for the error-prone cases. Workloads can be typically categorized into three categories: multiplication intensive, memory intensive and others. The detailed results are obtained by integrating error rates and error detection frequencies for one representative application belonging to each type - `dct`:multiplication intensive, `mcf`:memory-intensive, and `perlbench`:other.

In figure 6.10, the delay values are mapped to the z-axis and are normalized to those of TMR-AMT. Error injection rates are on the y-axis and error detection frequency of the system on the y-axis, both using a logarithmic scale. As mentioned before, the error detection frequency is varied between 10^2 and 10^6 instruction and the error rate between 10^{-4} and 10^{-9}. The surface in the figure makes up the delay values for various error rates. It can be observed that as the error detection frequency decreases, the delay value overhead of RRNS relative to TMR also decreases. This is attributed to the reduction in

the expensive error detection overhead of RRNS. In this sampling range, when the error rate value increases, the relative delay value also slightly increases, although the trend is not very straightforward from this figure. For HPC, 10^{-4} is already much higher than the expected error rate, so we are not necessary to evaluate the larger values. Using $(4, 2)$-RRNS, `perlbench`, `dct`, and `mcf` incur a delay overhead of 15.50%, 0.55%, and 8.23% respectively over TMR-AMT.

Figure 6.11 shows the energy consumption of RRNS normalized to TMR on the z-axis. The other two axes are similar to the ones in Figure 6.10. Energy consumption follows a similar trend to delay, and decreases with a decrease in error detection frequency. Moreover, energy increase with higher error injection rates is insignificant. `perlbench`, `dct`, and `mcf` consume 43.53%, 25.04%, and 44.67% of TMR's energy respectively. This implies energy savings of 56.47%, 74.96%, and 55.33% for each of the benchmarks. These significant savings are attributed to the comparatively lean bit-widths of the RRNS design when compared with the two normal threads' overhead in TMR. On average, using thread-level RRNS instead of TMR achieves 62.25% energy reduction.

Similarly, Figure 6.12 reveals the normalized results of the RRNS Energy Delay Product (EDP) on the Z-axis. The coordinate axes (x and y) and related parameters are the same as Figure 6.10 and Figure 6.11. The EDP of `perlbench`, `dct`, and `mcf` is 50.28%, 25.18%, and 48.53% that of TMR respectively. On average, using the proposed thread-level RRNS architecture reduces the EDP by 58.67% when replacing TMR. This reduction in EDP makes RRNS-AMT a great candidate for exascale systems where energy reduction with good performance characteristics are both important design criteria.

In this section, the evaluation results are based on an in-order RRNS core that supports the Asynchronous Many-Task programming model as proof of the great potential of RRNS in exascale computing systems where reliability and resiliency along with energy-efficiency are first world constraints. Future designs exploiting smarter pipelines may bring performance(delay) closer to traditional cores while providing the required characteristics

with negligible overheads.

6.5 Related Work

The Asynchronous Many-Task (AMT) programming model is a good candidate for exascale computing. Conventional bulk-synchronous programming models such as MIP-X [111] are unable to meet the critical requirements (such as performance heterogeneity, increase error mitigation, etc.) of exascale systems [28]. AMT is effectively able to alleviate these challenges, making it suitable for exascale computing [28]. The Habanero C/C++ library (HClib) [27] is one such AMT-based lightweight runtime that presents several programming constructs facilitating users to easily express parallelism. However, HClib initially did not directly target critical exascale computing issues such as fault-tolerance and energy efficiency. Resilience techniques in the form of task replication, task replay, Algorithm-Based Fault Tolerance (ABFT), and checkpointing were added as an extension to the runtime by [106]. Unlike other techniques, checkpointing works as a second-level technique such that if other techniques fail, the system can restart from an earlier checkpoint.

Task replication is similar to Dual/Triple Modular Redundancy [98] and directly replicates the entire computational logic. While easy to use since it requires no understanding of a specific algorithm and underlying application, the critical shortcoming of task replication is its extremely high energy overhead, with a theoretical overhead as high as 200%. On the other hand, task replay and ABFT require the programmer to define and implement specific error-detection algorithms, making them relatively difficult to use and a part of the white-box resilience techniques family. The RRNS based AMT model presented in this chapter overcomes these implementation challenges while maintaining its advantage of being extremely energy efficient.

On the software front, Chen et al. have proposed compiler-assisted resiliency methods in their work CARE [112]. However, CARE's error coverage rate is only around 83%,

much lower than the capability of RRNS, and hence it cannot guarantee a complete fault tolerance. Finally, CARE does not take the power consumption limitations of exascale computing into account. Its orthogonal features, when used in conjunction with RRNS can enhance the resiliency of the overall system.

Deng et al. [29, 30] have presented designs of unpipelined RRNS core microarchitectures that target upcoming millivolt switches [113, 114] with instruction behavior similar to Single Instruction Multiple Data (SIMD) architectures. In contrast, the design presented in this chapter is an in-order pipeline that can support AMT-based threading based on HClib. Moreover, this design's instruction execution mode is similar to Multiple Instruction Multiple Data (MIMD), allowing better system performance and making it more suitable for exascale systems.

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Conclusion

At around 2006, the transistors' energy density is no longer able to be described by Denard scaling. A critical reason for this Breakdown is the leakage current from the CMOS circuit, leading to an increase in energy consumption. Therefore, researchers began to design new millivolt switches to extend the technology roadmap scaling further. Next-generation devices such as tunneling FETs and ferroelectric FETs enable reducing the supply voltage to a few tens of millivolts. However, as a result of operating near the kT noise floor, they are subject to intermittent, stochastic bit errors *in logic*. Therefore, to use this new type of device, it is necessary to design energy-efficient and error-tolerant microarchitectures to extend Denard scaling further. RRNS is a promising approach towards using such ultra-low-power devices by employing efficient computational error correction and/or detection with checkpointing&restart.

This topic presents theoretical and practical aspects towards the design and analysis of scalable RRNS-based microarchitectures with correction and checkpointing&restart capabilities. Then, the RRNS-based static checkpointing&restart system is proposed and demonstrates its potential for further energy and EDP reduction. By combining the novel adaptive checkpointing&restart schemes, this scalable, resilient system design achieves notable energy reduction for non-memory-intensive workloads when compared to correction-only methodology, without significant impact on performance. Based on the error correction and error detection&restart mechanisms, the RRNS design space is also explored within a reasonable range to find the optimal or sub-optimal configuration points. After exploring RRNS design space systematically, *(4,2)-RRNS* configurations perform best in the

two-dimensional *(n,r)-RRNS* design plane. When compared to a binary core without computational resilience that runs at high V_{dd}, the proposed RRNS scalable microarchitecture reduces EDP by 53% on average for memory-intensive workloads and by 67% on average for non-memory-intensive workloads.

Since RRNS has great potential for energy efficiency and error tolerance, besides millivolt switches, RRNS can also be applied to exascale systems to alleviate their current critical challenges.

The system complexity of today's HPC continues to increase. Thus the probability of error also rapidly grows as more resources are integrated into the system. The Near Threshold Voltage (NTV) technique applied to the exascale system makes things even worse. In addition to the error rate, the exascale computer also faces the problem of power consumption restriction. Thus, more powerful and efficient fault-tolerant technologies are needed to ensure the entire system's correctness and improve its Mean Time Between Failures(MTBF). However, state-of-the-art resilient techniques for exascale computing only emphasize latency optimization and lack of power/energy considerations. According to the power consumption requirements of the exascale system proposed by DOE, the current most advanced exascale HPC still has a significant gap away from this threshold. Therefore, it is still necessary to find more efficient methods for power optimization. This thesis presents the thread-level RRNS concept and designs a microarchitecture with a corresponding Asynchronous Many-Task (AMT) programming framework that can efficiently support this scheme. Under the premise of sacrificing a small amount of speedup, the advantages of low energy consumption, better fault-tolerance, and ease of use are obtained. Finally, it can be concluded through experimental results that, on average, the thread-level RRNS method has an 8.23% increase in delay compared to TMR, but the related energy and EDP have 62.25% and 58.67% reductions, respectively.

7.2 Future Work

7.2.1 Energy and Speedup Tradeoff of Different Pipeline Designs

In the microarchitecture design for exascale computing, the in-order pipeline setting is used. This means that the instruction issue and complete totally follow the original program order. From an intuitive analysis, an out-of-order pipeline configuration requires a large number of extra hardware resources and operations, which should lead the system to become more power hungry. However, there are many different out-of-order execution versions, such as the scoreboard method, also known as FICO (Fire In-order Complete Out-of-order). It means that the instructions issue follows the original program order, but the sequence of instructions complete could be in any different order. Similarly, FOCO (Fire Out-of-order Complete Out-of-order) means that all the instructions could issue and complete in any order. Its order constraint should come from hazards, not the original program order. For example, Tomasulo Approach and PReg (RAT) Approach both belong to FOCO. One opportunity from this topic for future work is to design different versions of the pipeline microarchitecture with RRNS. It is also possible to build some compromise designs from existing approaches and make a better tradeoff between speedup and energy consumption. On this basis, it is possible to find out the most suitable RRNS pipeline design through experiments, which may further improve the system's reliability and energy benefits.

7.2.2 Error Model Improvement

The error probability of each instruction is accumulated to calculate the total number of errors, and subsequently the MTBF. This statistical error probability model can be further optimized by considering the error spreading behaviors within the system. The error model will be much closer to reality in terms of error occurrences by supporting the error spreading behaviors and further demonstrates RRNS techniques' benefits.

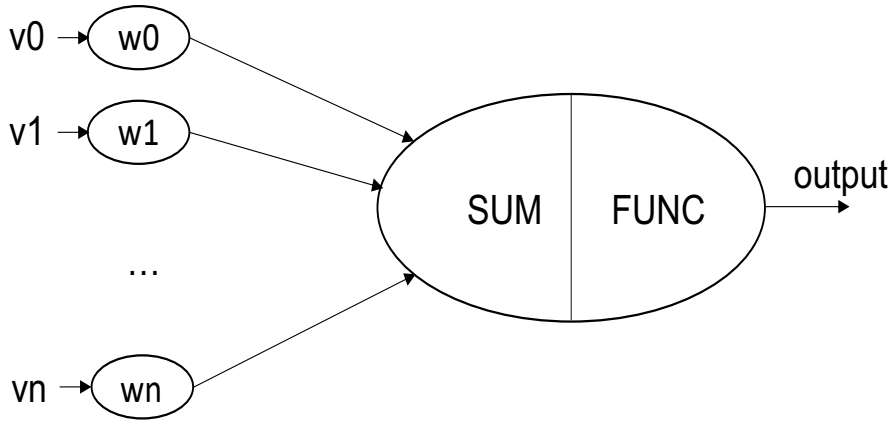


Figure 7.1: A Conventional Neuron of Deep Neural Network

7.2.3 RRNS Deep Neural Network

Deep Neural Network (DNN) [115] is currently a hot research domain, and it can be applied to a large number of applications [116]. Integrating RRNS into DNN may have a great potential to optimize the neutron structure and the entire network, thereby improving the reliability and achieving the purpose of energy efficiency.

Figure 7.1 is the basic structure of a neuron in DNN. Each neutron from each layer mainly includes three steps: 1) weight multiplications, 2) summation, and 3) executing an activation function. In other words, a large number of multiplications are required for neutron operations. The RRNS index-multiplication approach mentioned in Section 2.8 can also be applied here by further reducing the network's energy consumption and improving system reliability. Salamat et al. [20] use RNS in the neutron, but they did not consider the fault-tolerance problem and did not use the index-sum approach to optimize multipliers further.

A fault may be generated from inputs, connections, weights, multipliers, accumulators, and activation functions. Thus RRNS would be a good candidate for fault-tolerance in these scenarios. RRNS-DNN should change all the data representation from binary to RRNS, including inputs, weights, adders, multipliers, etc.

On the one hand, through the above discussion and analysis, RRNS-DNN contains

the following advantages: 1) Multiplication dominates DNN computing. RRNS works efficiently in both addition and multiplication. Moreover, RRNS index-sum multiplication (Replacing multiplication by one RRNS addition and two LUT accesses) makes this benefit from good to great. 2) Less RRNS unfriendly instructions, such as division and shifting. 3) The RRNS error detection does not need to be performed in every neuron. It only needs to check when it is necessary. Therefore, optimizing DNN through RRNS is not only possible to improve the network reliability but also likely to reduce energy consumption via simplifying the multiplier.

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] A. McMenamin, "The end of dennard scaling," 2013.

[3] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM journal of research and development 5.3*, pp. 183–191, 1961.

[4] M. Neyman, "The negentropy principle in information-processing systems," *Telecommunications and Radio Engineering 2*, 1966.

[5] U. E. Avci, D. H. Morris, and I. A. Young, "Tunnel field-effect transistors: Prospects and challenges," *IEEE Journal of the Electron Devices Society*, vol. 3, no. 3, pp. 88–95, 2015.

[6] S. George, K. Ma, A. Aziz, X. Li, A. Khan, S. Salahuddin, M. Chang, S. Datta, J. Sampson, S. Gupta, and V. Narayanan, "Nonvolatile memory design based on ferroelectric fets," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[7] S. Agarwal, J. Cook, E. DeBenedictis, M. P. Frank, G. Cauwenberghs, S. Srikanth, B. Deng, E. R. Hein, P. G. Rabbat, and T. M. Conte, "Energy efficiency limits of logic and memory," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, 2016, pp. 1–8.

[8] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[9] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[10] D. Lipetz and E. Schwarz, "Self checking in current floating-point units," in *2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 73–76.

[11] J. Warnock, Y. Chan, W. Huott, S. Carey, M. Fee, H. Wen, M. J. Saccamango, F. Malgioglio, P. Meaney, D. Plass, Y. H. Chan, M. Mayo, G. Mayer, L. Sigal, D. Rude, R. Averill, M. Wood, T. Strach, H. Smith, B. Curran, E. Schwarz, L. Eisen, D. Malone, S. Weitzel, P. K. Mak, T. McPherson, and C. Webb, "A 5.2ghz

microprocessor chip for the ibm zenterprise system," in *2011 IEEE International Solid-State Circuits Conference*, 2011, pp. 70–72.

[12] D. Henderson, B. Warner, and J. Mitchell, "Ibm power6 processor-based systems: Designed for availability," in *White Paper, IBM Corporation*, 2007.

[13] D. Henderson, J. Mitchell, and G. Ahrens, "Power7 system ras: Key aspects of power systems reliability, availability, and serviceability," in *White Paper, IBM Corporation*, 2010.

[14] R. Chokshi, K. S. Berezowski, A. Shrivastava, and S. J. Piestrak, "Exploiting residue number system for power-efficient digital signal processing in embedded processors," in *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, ACM, 2009, pp. 19–28.

[15] E. D. Di Claudio, F. Piazza, and G. Orlandi, "Fast combinatorial rns processors for dsp applications," *IEEE transactions on computers*, vol. 44, no. 5, pp. 624–633, 1995.

[16] J Ramirez, A Garcia, S Lopez-Buedo, and A Lloris, "Rns-enabled digital signal processor design," *Electronics Letters*, vol. 38, no. 6, pp. 266–268, 2002.

[17] J.-C. Bajard and L. Imbert, "A full rns implementation of rsa," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 769–774, 2004.

[18] C. Y. Hung and B. Parhami, "Fast rns division algorithms for fixed divisors with application to rsa encryption," *Information Processing Letters*, vol. 51, no. 4, pp. 163–169, 1994.

[19] S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon, "Rsa speedup with chinese remainder theorem immune against hardware fault cryptanalysis," *IEEE Transactions on computers*, vol. 52, no. 4, pp. 461–472, 2003.

[20] S. Salamat, M. Imani, S. Gupta, and T. Rosing, "Rnsnet: In-memory neural network acceleration using residue number system," in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, 2018, pp. 1–12.

[21] E. B. Olsen, "Introduction of the residue number arithmetic logic unit with brief computational complexity analysis (rez-9 soft processor)," *Whitepaper, Digital System Research*, 2015.

[22] D. Anderson, "Design and implementation of an instruction set architecture and an instruction execution unit for the rez9 coprocessor system," *M.S. Thesis, U of Nevada LV*, 2014.

[23] M. Labafniya and M. Eshghi, "Non-iterative rns division algorithm," 2012.

[24] S Talahmeh and P Siy, "Arithmetic division in rns using galois field gf(p)," *Computers and Mathematics with Applications*, vol. 39, no. 5, pp. 227 –238, 2000.

[25] R. W. Watson, "Error detection and correction and other residue interacting operations in a residue redundant number system," in *Univ. California, Berkeley*, 1965.

[26] B. Deng, S. Srikanth, A. Jain, T. Conte, E. Debenedictis, and J. Cook, "Scalable energy-efficient microarchitectures with computational error tolerance via redundant residue number systems," *IEEE Transactions on Computers*, Early Access, 2021.

[27] M. Grossman, V. Kumar, N. Vrvilo, Z. Budimlic, and V. Sarkar, "A pluggable framework for composable hpc scheduling libraries," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 723–732.

[28] J. Wilke, K. Franko, D. Hollman, S. Knight, H. Kolla, P. Lin, G. Sjaardema, N. Slattengren, K. Teranishi, J. Bennett, and R. Clay, *Asynchronous many-task programming models for next generation platforms*, URL: `https://www.osti.gov/biblio/1261059`, 2015.

[29] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook, "Computationally-redundant energy-efficient processing for y'all (creepy)," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, 2016, pp. 1–8.

[30] B. Deng, S. Srikanth, E. R. Hein, T. M. Conte, E. Debenedictis, J. Cook, and M. P. Frank, "Extending moore's law via computationally error-tolerant computing," *ACM Trans. Archit. Code Optim (TACO).*, vol. 15, no. 1, 8:1–8:27, Mar. 2018.

[31] S. Srikanth, P. G. Rabbat, E. R. Hein, B. Deng, T. M. Conte, E. DeBenedictis, J. Cook, and M. P. Frank, "Memory system design for ultra low power, computationally error resilient processor microarchitectures," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 696–709.

[32] A. Omondi and B. Premkumar, "Residue number systems: Theory and implementation," Imperial College Press.

[33] C. W. Hastings, "Automatic detection and correction of errors in digital computers using residue arithmetic," in *Region Six Annu. Conf.*, IEEE, 1966, pp. 429–464.

[34] O. Goldreich, D. Ron, and M. Sudan, "Chinese remaindering with errors," in *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, ACM, 1999, pp. 225–234.

[35] R. W. Watson and C. W. Hastings, "Self-checked computation using residue arithmetic," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1920–1931, 1966.

[36] L. liang Yang and L. Hanzo, "Coding theory and performance of redundant residue number system codes," *IEEE TRANS. INFORM. THEORY*, 1999.

[37] J.-S. Chiang and M. Lu, "Floating-point numbers in residue number systems," *Computers & Mathematics with Applications*, vol. 22, no. 10, pp. 127–140, 1991.

[38] E. G. Walters III, M. G. Arnold, and M. J. Schulte, "Using truncated multipliers in dct and idct hardware accelerators," in *Optical Science and Technology, SPIE's 48th Annual Meeting*, International Society for Optics and Photonics, 2003, pp. 573–584.

[39] A. Preethy and D Radhakrishnan, "A 36-bit balanced moduli mac architecture," in *Circuits and Systems, 1999. 42nd Midwest Symposium on*, IEEE, vol. 1, 1999, pp. 380–383.

[40] A. Preethy and D. Radhakrishnan, "Rns-based logarithmic adder," in *IEE Proceedings-Computers and Digital Techniques*, vol. 147, IET, 2000, pp. 283–287.

[41] B. P. Yinan Kong, *Residue number system scaling schemes*, 2005.

[42] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proceedings of the 18th Annual International Conference on Supercomputing*, ser. ICS '04, Malo, France: ACM, 2004, pp. 277–286, ISBN: 1-58113-839-3.

[43] T. F. Tay and C.-H. Chang, "A non-iterative multiple residue digit error detection and correction algorithm in rrns," *IEEE transactions on computers*, vol. 65, no. 2, pp. 396–408, 2016.

[44] J.-C. Bajard, J. Eynard, and N. Merkiche, "Multi-fault attack detection for rns cryptographic architecture," in *Computer Arithmetic (ARITH), 2016 IEEE 23nd Symposium on*, IEEE, 2016, pp. 16–23.

[45] H. Xiao, H. K. Garg, J. Hu, and G. Xiao, "New error control algorithms for residue number system codes," *ETRI Journal*, vol. 38, no. 2, pp. 326–336, 2016.

[46] L. Xiao and X.-G. Xia, "Error correction in polynomial remainder codes with non-pairwise coprime moduli and robust chinese remainder theorem for polynomials," *IEEE Transactions on Communications*, vol. 63, no. 3, pp. 605–616, 2015.

[47] C.-H. Chang, A. S. Molahosseini, A. A. E. Zarandi, and T. F. Tay, "Residue number systems: A new paradigm to datapath optimization for low-power and high-performance digital signal processing applications," *IEEE circuits and systems magazine*, vol. 15, no. 4, pp. 26–44, 2015.

[48] T. F. Tay and C.-H. Chang, "A new algorithm for single residue digit error correction in redundant residue number system," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, IEEE, 2014, pp. 1748–1751.

[49] P. Yin and L. Li, "A new algorithm for single error correction in rrns," in *Communications, Circuits and Systems (ICCCAS), 2013 International Conference on*, IEEE, vol. 2, 2013, pp. 178–181.

[50] H.-Y. Lo and T.-W. Lin, "Parallel algorithms for residue scaling and error correction in residue arithmetic," *Wireless Engineering and Technology*, vol. 4, no. 04, p. 198, 2013.

[51] A. Sengupta and B. Natarajan, "Performance of systematic rrns based space-time block codes with probability-aware adaptive demapping," *IEEE Transactions on Wireless Communications*, vol. 12, no. 5, pp. 2458–2469, 2013.

[52] N. Z. Haron and S. Hamdioui, "Redundant residue number system code for fault-tolerant hybrid memories," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 7, no. 1, p. 4, 2011.

[53] Y. Tang, E. Boutillon, C. Jégo, and M. Jézéquel, "A new single-error correction scheme based on self-diagnosis residue number arithmetic," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, IEEE, 2010, pp. 27–33.

[54] V. T. Goh and M. U. Siddiqi, "Multiple error detection and correction based on redundant residue number systems," *IEEE Transactions on Communications*, vol. 56, no. 3, 2008.

[55] A. Sweidan and A. A. Hiasat, "On the theory of error control based on moduli with common factors," *Reliable computing*, vol. 7, no. 3, pp. 209–218, 2001.

[56] R. S. Katti, "A new residue arithmetic error correction scheme," *IEEE transactions on computers*, vol. 45, no. 1, pp. 13–19, 1996.

[57] H. Krishna, B. Krishna, K.-Y. Lin, and J.-D. Sun, *Computational Number Theory and Digital Signal Processing: Fast Algorithms and Error Control Techniques*. CRC Press, 1994, vol. 6.

[58] E. D. Di Claudio, G. Orlandi, and F. Piazza, "A systolic redundant residue arithmetic error correction circuit," *IEEE Transactions on Computers*, vol. 42, no. 4, pp. 427–432, 1993.

[59] H. Krishna, K.-Y. Lin, and J.-D. Sun, "A coding theory approach to error control in redundant residue number systems. i. theory and single error correction," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 8–17, 1992.

[60] J.-D. Sun and H. Krishna, "A coding theory approach to error control in redundant residue number systems. ii. multiple error detection and correction," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 39, no. 1, pp. 18–34, 1992.

[61] J.-D. Sun, H. Krishna, and K. Lin, "A superfast algorithm for single-error correction in rrns and hardware implementation," in *Circuits and Systems, 1992. ISCAS'92. Proceedings., 1992 IEEE International Symposium on*, IEEE, vol. 2, 1992, pp. 795–798.

[62] G. A. Orton, L. E. Peppard, and S. E. Tavares, "New fault tolerant techniques for residue number systems," *IEEE transactions on computers*, vol. 41, no. 11, pp. 1453–1464, 1992.

[63] C.-C. Su and H.-Y. Lo, "An algorithm for scaling and single residue error correction in residue number systems," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1053–1064, 1990.

[64] V. Ramachandran, "Single residue error correction in residue number systems," *IEEE transactions on computers*, vol. 32, no. 5, pp. 504–507, 1983.

[65] M Etzel and W Jenkins, "Redundant residue number systems for error detection and correction in digital filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 5, pp. 538–545, 1980.

[66] F. Barsi and P. Maestrini, "Error detection and correction by product codes in residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 915–924, 1974.

[67] S.-S. Yau and Y.-C. Liu, "Error correction in redundant residue number systems," *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 5–11, 1973.

[68] T. R. Rao, "Biresidue error-correcting codes for computer arithmetic," *IEEE Transactions on computers*, vol. 100, no. 5, pp. 398–402, 1970.

[69] N. S. Szabo and R. I. Tanaka, *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[70] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Microarchitecture,MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, IEEE, 2003, pp. 7–18.

[71] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *High Performance Computer Architecture, HPCA, IEEE 14th International Symposium on*, IEEE, 2008, pp. 381–392.

[72] T. M. Austin, "Diva: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture, MICRO-32. Proceedings. 32nd Annual International Symposium on*, IEEE, 1999, pp. 196–207.

[73] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, 6:1–6:29, Jun. 2011.

[74] J. S. Plank, Kai Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.

[75] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.

[76] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, Feb. 2006.

[77] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 129–144.

[78] G. Levitin, L. Xing, Y. Dai, and V. M. Vokkarane, "Dynamic checkpointing policy in heterogeneous real-time standby systems," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1449–1456, 2017.

[79] I. Akturk and U. R. Karpuzcu, "Acr: Amnesic checkpointing and recovery," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 30–43.

[80] K. B. Ferreira, R. Riesen, R. Brighwell, P. Bridges, and D. Arnold, "Libhashckpt: Hash-based incremental checkpointing using gpuâs," in *European MPI Users' Group Meeting*, Springer, 2011, pp. 272–281.

[81] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *2012 41st International Conference on Parallel Processing*, 2012, pp. 148–157.

[82] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[83] S. Rusu, "Multi-domain processors design overview," in *ISCA tutorial on Multi-domain Processors: Challenges, Design Methods, and Recent Developments*, 2010.

[84] Y. Shimazaki, R. Zlatanovici, and B. Nikolic, "A shared-well dual-supply-voltage 64-bit alu," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 3, pp. 494–500, 2004.

[85] S. K. Samal, S. Khandelwal, A. I. Khan, S. Salahuddin, C. Hu, and S. K. Lim, "Full chip power benefits with negative capacitance fets," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2017, pp. 1–6.

[86] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "An analysis of resilience techniques for exascale computing platforms," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 914–923.

[87] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kalé, "Using migratable objects to enhance fault tolerance schemes in supercomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 2061–2074, 2015.

[88] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, "Near-threshold voltage (ntv) design — opportunities and challenges," in *DAC Design Automation Conference 2012*, 2012, pp. 1149–1154.

[89] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 2012, pp. 615–626.

[90] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 180, 2009, p. 012 045.

[91] T. Scogland, B. Subramaniam, and W.-C. Feng, "The green500 list: Escapades to exascale," *Comput. Sci.*, vol. 28, no. 2–3, 109–117, May 2013.

[92] S. Labasan, "Energy-efficient and power-constrained techniques for exascale computing," *Semanticscholar: Seattle, WA, USA*, 2016.

[93] "Doe exascale initiative roadmap," in *Architecture and Technology Workshop*, 2009.

[94] S. A. et al., "The opportunities and challenges of exascale computing," in *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, 2010.

[95] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, vol. 15, 2008.

[96] *Frontier (olcf-5) - supercomputers*, URL: https://en.wikichip.org/wiki/supercomputers/frontier.

[97] A. Geist, *Ornl's frontier exascale computer*, URL: https://smc.ornl.gov/wp-content/uploads/2019/09/Geist-presentation-2019.pdf, 2019.

[98] J. V. Neumann, "Probabilistic logics and the synthesis of reliableorganisms from unreliable components," in *Automata studies*, vol. 34, 1956, 43–98.

[99] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[100] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers and Innovations: an International Journal*, vol. 1, no. 1, pp. 5–28, 2014.

[101] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," ser. LCTES'15, Association for Computing Machinery, 2015, ISBN: 9781450332576.

[102] W. D. Gropp, E. L. Lusk, and A. Skjellum, *Using mpi - portable parallel programming with the message-parsing interface*, URL: https://www.worldcat.org/oclc/41548279, 1999.

[103] N. Gupta, J. R. Mayo, A. S. Lemoine, and H. Kaiser, "Towards distributed software resilience in asynchronous many-task programming models," in *2020 IEEE/ACM*

10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), IEEE, 2020, pp. 11–20.

[104] O. Subasi, J. Arias, O. Unsal, J. Labarta, and A. Cristal, "Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 99–102.

[105] C. Cao, T. Herault, G. Bosilca, and J. Dongarra, "Design for a soft error resilient dynamic task-based runtime," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 765–774.

[106] S. R. Paul, A. Hayashi, N. Slattengren, H. Kolla, M. Whitlock, S. Bak, K. Teranishi, J. Mayo, and V. Sarkar, "Enabling resilience in asynchronous many-task programming models," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed., Cham: Springer International Publishing, 2019, pp. 346–360.

[107] S. R. Paul, A. Hayashi, M. Whitlock, S. Bak, K. Teranishi, J. Mayo, M. Grossman, and V. Sarkar, "Integrating inter-node communication with a resilient asynchronous many-task runtime system," in *2020 Workshop on Exascale MPI (ExaMPI)*, 2020, pp. 41–51.

[108] *Future*, URL: `http://www.cplusplus.com/reference/future/`.

[109] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[110] L.-L. Yang and L. Hanzo, "Coding theory and performance of redundant residue number system codes," *IEEE Trans. Inform. Theory*, 1999.

[111] *Compilers and more: Mpi+x*, URL: `https://www.hpcwire.com/2014/07/16/compilers-mpix/`.

[112] C. Chen, G. Eisenhauer, S. Pande, and Q. Guan, "Care: Compiler-assisted recovery from soft failures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[113] U. E. Avci, D. H. Morris, and I. A. Young, "Tunnel field-effect transistors: Prospects and challenges," *IEEE Journal of the Electron Devices Society*, vol. 3, no. 3, pp. 88–95, 2015.

[114] S. George, K. Ma, A. Aziz, X. Li, A. Khan, S. Salahuddin, M. Chang, S. Datta, J. Sampson, S. Gupta, and V. Narayanan, "Nonvolatile memory design based on

ferroelectric fets," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[115] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[116] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of big data*, vol. 2, no. 1, pp. 1–21, 2015.